31ème Journées Francophones des Langages Applicatifs

Zaynah Lea Dargaye, Yann Regis-Gianas

31ème Journées Francophones des Langages Applicatifs

du 29 janvier au 1er février 2020 à Gruissan

Préface

Les JFLA réunissent chaque année, dans un cadre convivial, concepteurs, développeurs et utilisateurs des langages fonctionnels, des assistants de preuve et des outils de vérification de programmes. Le spectre des travaux présentés aux JFLA est très large : il touche ainsi les aspects les plus théoriques de la conception des langages jusqu'à ses applications industrielles. Après avoir fêté sa trentième édition dans le Jura, les JFLA se déroulent cette année à la mer, et plus précisément au bord du Golf du Lion, en Occitanie, près de Narbonne, à l'hôtel "Phoebus Garden & Spa" de Gruissan.

Cette année, nous avons sélectionné 8 articles de recherche et 9 articles courts. Le programme de cette année aborde des thématiques variées, des fondements sémantique de la logique linéaire différentielle finitaire polarisée à la sûreté de code assembleur intégré dans du code C. Ce programme met en avant l'intérêt grandissant de la communauté des JFLA pour le développement de méthodes formelles destinées à l'ingénierie de la preuve au travers de plateformes logiques pour le raisonnement sur les sémantiques, la génération de certificats logiques pour les transformations d'outils formels, l'amélioration de la part d'automatisation dans les assistants de preuve, la vérification de bibliothèques utilisées par les solveurs de contraintes ou bien encore la formalisation des nombres réels garantissant la terminaison des calculs. Ce programme témoigne aussi de l'usage de la preuve formelle pour montrer la sûreté de systèmes variés tels que les bases de données ou bien encore les textes législatifs régissant l'impôt sur le revenu.

Les JFLA présentent traditionnellement des travaux sur la conception des langages de programmation et de leurs outils. L'édition 2020 poursuit cette tradition en proposant tout d'abord des travaux à fins pédagogiques que cela soit pour éviter les débordements de pile en programmation OCaml ou bien pour améliorer la qualité des outils d'apprentissage de ce langage de programmation. Sur ce thème, ce programme offre aussi des présentations d'outils formels d'assistance au développement que ce soit pour détecter des redondances de code ou encore pour garantir statiquement de bonnes propriétés d'un glaneur de cellules. Finalement, nous découvrirons des langages pour la conception, la preuve et l'implémentation de programmes probabilistes et aussi un codage monadique pour l'implémentation sûre de programmes globalement asynchrones et localement synchrones.

Cette sélection a été faite par les membres du comité de programme, que nous remercions chaleureusement, à partir des 20 soumissions. Nous tenons à exprimer nos sincères remerciements aux rapporteurs externes au comité de programme : Kenji Maillard et Samuel Mimram.

En plus du programme sélectionné, nous aurons le grand plaisir d'écouter deux cours présentant des outils de vérifications formelles. Tout d'abord, Sylvain Conchon nous parlera de "Cubicle, vérification de modèles pour des propriétés de sûreté des systèmes basés sur les tableaux". Pour le second cours, Claire Dross présentera "SPARK 2014 : la Preuve de programme pour les développeurs". Deux exposés invités compléteront enfin ce programme. Hugo Herbelin nous parlera des directions futures possibles pour la logique de Coq tandis que Pierre-Évariste Dagand nous parlera de l'expérience Usuba, un compilateur post-Moore.

Nous tenons à remercier ici celles sans qui nous n'aurions pu organiser les JFLA 2020 : Frédérique Descreaux, assistante du département DILS au CEA List qui nous a guidé pour toute la logistique et Emmanuelle Dauvergne, gestionnaire financier des projets du DILS au CEA List. Enfin, nos sincères remerciements à nos généreux sponsors. Cette année encore, les étudiant.e.s orateur.e.s ne paient pas les frais d'hébergement et d'inscription. Merci à Nomadic Labs, au CEA List, à OCamlPro, à Tarides, à Tweag.io, à la fondation OCaml, au GDR GPL et à TrustInSoft.

Zaynah Dargaye et Yann Régis-Gianas.

Comité de programme

Valentin Blot	Inria Saclay
Vincent Botbol	Nomadic Labs
Emmmanuel Chailloux	LIP6 - Sorbonne Université
Zaynah Dargaye	Nomadic Labs
Delphine Demange	Univ Rennes, Inria, CNRS, IRISA
Chantal Keller	LRI, Université Paris-Sud
Marie Kerjean	Inria Rennes-Bretagne Atlantique
Alain Mebsout	OCamlPRO
Julien Narboux	ICube Université de Strasbourg
Marie Pelleau	Université de Nice
Pierre-Marie Pédrot	Inria Rennes-Bretagne Atlantique
Yann Regis-Gianas	IRIF, Inria Paris, Université de Paris
Gabriel Scherer	Inria Saclay

Rapporteurs externes

Maillard, Kenji
Mimram, Samuel

Table des matières

Théorie des types cubiques, égalité polymorphe ad hoc et paramétricité itérée

Hugo Herbelin

La *théorie des types homotopique* [14] revisite la théorie des types de Martin-Löf sous l'angle de la correspondance observée au milieu des années 2000 entre égalité et chemins d'un espace topologique à homotopie près [15, 5], y introduisant 1) une généralisation de la correspondance de Curry-Howard à travers le concept unificateur de niveau d'homotopie 2) une nouvelle notion séduisante de types quotients appelée « types inductifs supérieurs » 3) dans la lignée de Hofmann et Streicher, une notion intuitive d'extensionnalité pour les types dénommée « univalence » [11, 15].

La théorie des types de Martin-Löf – tout comme sa variante imprédicative que Coq implémente [10] – est une logique, mais aussi un langage de programmation typé qui calcule. La question de comment calculer avec l'univalence et avec les types inductifs supérieurs s'est alors posée. Développée par Coquand et ses coauteurs, la *théorie des types cubique* [8, 9, 4] donne une réponse à cette question.

Un ingrédient clé de la théorie des types cubique est l'égalité cubique. D'inspiration géométrique, l'égalité cubique caractérise la prouvabilité de l'égalité de t et u de type A par l'existence d'une fonction f d'un intervalle abstrait $\mathbb{I} \triangleq [0;1]$ vers A telle que $f(0) \equiv t$ et $f(1) \equiv u$. Le mot « abstrait » indique ici que $\mathbb{I}$ n'est équipé d'aucune structure analytique : la seule information que l'on ait de $\mathbb{I}$ est qu'il contient deux éléments nommés 0 et 1 reliés symboliquement par un segment. Si i est une variable dans l'intervalle et p une preuve d'égalité entre t et u de type A, on pourra alors construire un point $p\,i$ flottant quelque part entre $p\,0 \equiv t$ et $p\,1 \equiv u$. Inversement, si p_i est un point de type A potentiellement dépendant d'une variable d'intervalle i, l'abstraction $\lambda i.p_i$ représentera une preuve de l'égalité des valeurs prises par p aux extrémités de l'intervalle, c'est-à-dire une preuve de l'égalité $p_0 =_A p_1$.

L'égalité cubique se généralise en une égalité hétérogène permettant de comparer t de type A et u de type B sous l'hypothèse que A et B sont eux-même égaux en tant que types. Par exemple, si t est un vecteur de taille $n+p$ et u un vecteur de taille $p+n$, t et u sont comparables relativement à la propriété de commutation de l'addition. En pratique, l'égalité cubique a ainsi la souplesse de l'égalité hétérogène dite de John Major [12] mais sans l'inconvénient de forcer la logique ambiante à trivialiser le contenu calculatoire des preuves d'égalité. On peut alors parler d' « égalité au dessus de » et une propriété importante de cette égalité est qu'elle peut s'itérer, caractérisant alors des carrés, des cubes, des hypercubes d'égalités, etc. d'où le nom aussi d'égalité cubique.

Un deuxième ingrédient de la théorie des types cubiques est le basculement d'un contenu calculatoire générique de l'égalité tel qu'on peut le voir dans la théorie des types de Martin-Löf vers un contenu calculatoire ad hoc tel qu'on peut le voir dans la théorie des types observationnelle de Altenkirch et ses coauteurs [1, 3]. Cet aspect permet de capturer l'extensionnalité de l'égalité, et en particulier l'extensionnalité des fonctions et l'extensionnalité des types.

Dans cet exposé, nous ferons une relecture de la théorie des types cubique comme style direct pour une traduction de paramétricité, explorant un cran plus loin comment l'égalité peut être simultanément traitée sous une forme abstraite permettant le raisonnement déductif (égalité comme chemin, équipée avec réflexivité et congruence) et une vision polymorphe ad hoc caractérisant l'égalité extensionnellement par induction sur la structure du type. La paramétricité que nous considèrerons sera une forme itérée telle qu'on peut la trouver chez Bernardy et Moulin [7, 6] d'une paramétricité univalente telle qu'on peut la trouver chez Tabareau, Tanter et Sozeau [13] ou chez Altenkirch et Kaposi [2]. La substitutivité de l'égalité arrivera alors comme conséquence de l'extensionnalité de l'égalité sur les types.

L'exposé sera basé sur des travaux réalisés en commun avec Hugo Moeneclaey.

Références

[1] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 412–420. IEEE Computer Society, 1999.

[2] Thorsten Altenkirch and Ambrus Kaposi. Towards a cubical type theory without an interval. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 3 :1–3 :27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[3] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now ! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007.

[4] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. Cartesian cubical type theory. On ArXiv, 2017.

[5] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1) :45–55, Jan 2009.

[6] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electr. Notes Theor. Comput. Sci.*, 319 :67–82, 2015.

[7] Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 135–144, Washington, DC, USA, 2012. IEEE Computer Society.

[8] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 107–128. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[9] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory : A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *TYPES 2015*, volume 69 of *LIPIcs*, pages 5 :1–5 :34. Schloss Dagstuhl, 2018.

[10] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[11] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 208–212. IEEE Computer Society, 1994.

[12] Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.

[13] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free : univalent parametricity for effective transport. *PACMPL*, 2(ICFP) :92 :1–92 :29, 2018.

[14] The Univalent Foundations Program. *Homotopy Type Theory Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. http://homotopytypetheory.org/book/.

[15] Vladimir Voevodsky. A very short note on the homotopy λ-calculus. http ://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf, 2006.

Cubicle : Model Checking Modulo Théories

Sylvain Conchon

Résumé

Ce cours est une introduction au model checking modulo théories (MCMT), une technique de vérification de modèles inventée par Silvio Ghilardi et Silvio Ranise.

Cette technique repose sur une extension d'un algorithme d'atteignabilité arrière qui intègre un démonstrateur automatique SMT.

Nous illustrerons les concepts de MCMT avec Cubicle, un vérificateur de modèle open source conçu pour vérifier des systèmes de transition basés tableaux.

Il s'agit d'une classe de systèmes paramétrés dont les états sont représentés par des tableaux indexés par des identificateurs de processus.

Ce cours présentera les dernières évolutions de Cubicle, en particulier celles permettant d'analyser des systèmes de transition avec mémoires faibles.

The Usuba Experiment
A Compiler in a post-Moore World

Darius Mercadier Pierre-Évariste Dagand[†]

Cnrs – Inria Paris – Sorbonne Université – LIP6

Abstract

Cryptographic primitives are subject to diverging imperatives. Functional correctness and auditability pushes for the use of a high-level programming language. Performance and the threat of timing attacks push for using no more abstract than an assembler to exploit (or avoid!) the micro-architectural features of a given machine. We believe that a suitable programming language can reconcile both views and actually improve on the state of the art of both.

Usuba is an opinionated dataflow programming language in which block ciphers become so simple as to be "obviously correct" and whose types document and enforce valid parallelization strategies at the granularity of individual bits. Its optimizing compiler, `usubac`, produces high-throughput, constant-time implementations performing on par with hand-tuned reference implementations. The cornerstone of our approach is a systematization and generalization of *bitslicing*, an implementation trick frequently used by cryptographers.

We thus show that Usuba can produce code that executes between 5% slower to 22% faster than hand-tuned reference implementations while gracefully scaling across a wide range of architectures and automatically exploiting Single Instruction Multiple Data (SIMD) instructions whenever the cipher's structure allows it.

This project also serves a crash course in computational survivalism and a training ground for the End of Moore's Law. We are indeed firmly in an era where performance gains are made by identifying high-level abstractions and mapping them carefully to specialized hardware. We will touch upon the challenges and opportunities faced by language designers and compiler writers in such a diverse, yet unforgiving, ecosystem.

SPARK 2014 : La Preuve de Programme pour les Développeurs

Claire Dross
AdaCore, Paris
dross@adacore.com

Abstract

Ce cours présente le langage SPARK, ainsi que l'outil de vérification statique lui étant associé. La preuve de programme et le développement logiciel ont beaucoup de similitudes, et gagnent à être menés de concert. L'accent sera mis dans ce cours sur l'utilisation pratique de l'outil de preuve à destination des développeurs.

1 Le langage SPARK

SPARK* est un langage de programmation basé sur le langage Ada. Il est associé à un ensemble d'outils de vérification visant à permettre le développement de logiciel à haut niveau de criticité.

Le langage Ada, créé dans les années 1980, est un langage impératif généraliste de style procédural. Il supporte la programmation orientée objet ainsi que la programmation concurrente nativement. Ada a été conçu pour le développement de logiciel critique. En conséquence, il a un certain nombre de particularités qui se prêtent bien à l'analyse statique. Tout d'abord, Ada est fortement typé, ce qui veut dire qu'à chaque expression est associé un et un seul type, et que toute conversion doit être explicite. Le système de type de Ada est assez riche. Il permet en particulier d'associer des contraintes supplémentaires à certains types, des intervalles sur les types d'entiers par exemple. La vérification de ces contraintes est imposée par le langage, soit à la compilation, si l'expression est connue statiquement, soit à l'exécution. Dans ce second cas, une exception est lancée en cas de violation. Parmi les autres vérifications imposées par Ada, on trouve l'absence de division par zéro, de débordement lors d'un accès dans un tableau, ou de déréférencement d'un pointeur nul. La dernière version du langage, Ada 2012, permet l'écriture de contrats, pré et postconditions sur les fonctions, et invariant de types. Ces contrats sont tout d'abord destinés à être vérifiés dynamiquement. Si les assertions sont activées lors de la compilation, une exception sera levée pour toute violation d'un contrat lors de l'exécution du programme.

* https://www.adacore.com/about-spark

SPARK est un sous-ensemble de Ada. Il restreint ou exclut les caractéristiques du langage ne se prêtant pas à l'analyse statique. En particulier, il ne permet pas de rattraper des exceptions, et il interdit les effets de bord dans les expressions ainsi que l'aliasing d'objets mutables.

2 L'outil GNATprove

GNATprove permet d'effectuer plusieurs types d'analyse. Premièrement, il vérifie que le programme respecte les contraintes imposées par le langage SPARK. Ensuite, grâce à une analyse de flot de données, il calcule l'ensemble des variables globales auxquelles chaque fonction accède, et il s'assure que le programme ne lit aucune valeur non initialisée. Finalement, il vérifie par analyse déductive toutes les contraintes dynamiques associées au langage (intervalle d'entiers, division par zéro, accès dans un tableau…) ainsi que tous les contrats fonctionnels et invariants de type. Cette dernière analyse est effectuée grâce à une traduction vers le langage d'entrée de l'outil Why3 (Bobot, Filliâtre, Marché, & Paskevich, 2011) qui génère des formules logiques correspondant à chacune des vérifications de SPARK par un calcul de plus faible précondition. Ces formules logiques, appelées obligations de preuve, sont ensuite envoyées vers des prouveurs automatiques.

Cette dernière analyse, la plus puissante, permet de vérifier des propriétés fonctionnelles complexes sur les programmes. Toutefois, elle nécessite pour fonctionner un effort d'annotation qui peut parfois être important, ainsi qu'une certaine maîtrise de l'outil pour comprendre d'où viennent les erreurs remontées. SPARK et GNATprove offrent un certain nombre de caractéristiques pouvant aider dans ce domaine. Nous verrons la génération de contre-exemples, le code fantôme, la librairie de lemmes mathématiques ainsi que l'interface de preuve interactive.

References

Bobot, F., Filliâtre, J.-C., Marché, C., & Paskevich, A. (2011). Why3: Shepherd your herd of provers.

Chiralités et exponentielles : un peu de différentiation

Esaïe Bauer[1] and Marie Kerjean[2]

[1] ENS Lyon
[2] LS2N et Inria, Équipe Gallinette

Résumé

Nous donnons une sémantique catégorique à la logique linéaire différentielle finitaire et polarisée. Cette axiomatique donne un cadre à des modèles polarisés dans les espaces vectoriels topologiques. Elle s'appuie sur la notion de chiralités et les modèles non-polarisés de la logique linéaire différentielle munis d'un biproduit.

1 Introduction

L'informatique, comme la physique et la biologie, est source de structures abstraites et de modèles dont l'étude est intrinsèquement intéressante. Ces différentes perspectives sémantiques - entre modèles d'un mouvement physique et modèle d'un calcul - se rejoignent parfois. En particulier, on retrouve dans l'étude de certains modèles dénotationnels du lambda-calcul une caractérisation des preuves en terme de *linéarité* des fonctions. La logique linéaire (LL) [8] désigne le calcul des séquent extrait de ce modèle, et permet une compréhension des règles logiques structurelles en terme de gestion des ressources. De même, l'interprétation des formules de LL par des espaces vectoriels permet de donner un contenu logique à la notion de différentielle. La logique linéaire différentielle (DiLL) [3] est un calcul des séquents qui rajoute ainsi à LL des règles co-structurelles, décrivant la différentiation des fonctions d'ordres supérieures.

La logique linéaire (LL) construit un transfert de techniques entre l'algèbre et la logique : elle permet un nouveau point de vue sur le tiers-exclu, et la règle d'élimination de la double négation qui en découle : $\neg\neg A \to A$. LL raffine la logique intuitionniste en une logique classique *pour une notion linéaire de négation*. Quand en logique classique la négation d'une formule A est équivalent à l'implication de l'absurde par la formule A :

$$\neg A \equiv A \Rightarrow \bot,$$

en logique linéaire la négation correspond naturellement à l'implication linéaire du faux par la formule :

$$A^{\perp} \equiv A \multimap \bot.$$

Cette nuance a permis historiquement d'introduire la notion de *dualité* et de *polarisation* en théorie de la preuve. En particulier, la polarisation distingue deux classes de formules, négatives et positives. Les formules négatives sont celles qui sont par construction stables par double négation, alors que les formules positives se comprennent comme celles à qui un défaut de structure ne permet pas cette invariance [9]. Cette distinction induite par la notion de négation linéaire trouve ensuite des applications en recherche de preuve [1].

Alors que la sémantique dénotationnelle s'attache à interpréter les programmes comme des fonctions – principalement dans la théorie des domaines, la sémantique catégorique est plus spécifique et interprète les formules comme des catégories munis de lois spécifiques. La théorie des catégories se place en particulier dans la continuité de la correspondance de Curry-Howard. Un programme *fonctionnel typé* de type $A \Rightarrow B$ est interprété comme une preuve de la formule B sous l'hypothèse A. A leur tour, ces preuves de $A \vdash B$ sont interprétées comme des morphismes de l'objet A vers l'objet B dans une certaine catégorie.

Dans cet article, nous nous intéressons aux modèles catégoriques de DiLL, c'est à dire aux axiomes sur une ou plusieurs catégories qui permettent d'y interpréter DiLL de manière invariante par élimination des coupures.

La différentiation et la notion de polarité n'ont à priori rien à voir : quand la différentielle est historiquement un objet intervenant en analyse, la polarité appartient purement à la théorie des langages de programmation. Pourtant, la dualité au sens algébrique du terme [1] est au centre de la théorie des distributions [19] et d'une majeure partie

1. la dualité en algèbre linéaire et en analyse fonctionnelle décrit l'interaction entre un $\mathbb{K}$-espace vectoriel E et l'ensemble des formes linéaires (continues) sur cet espace $E' := \mathcal{L}(E, \mathbb{K})$.

de l'analyse fonctionelle [11]. Mais lorsqu'on cherche à interpréter DiLL en analyse, on se heurte pourtant à des contraintes de polarité [13]. *L'injonction d'une négation linéaire involutive est trop contraignante* pour permettre à la fois de manipuler des différentielles et pour interpréter toutes les formules de la logique linéaire différentielle dans une même catégorie. Les premiers modèles qui apparaissent sont donc naturellement polarisés : les formules de DiLL sont interprétées par deux classes duales d'espaces topologiques.

La logique linéaire différentielle polarisée, déjà explorée par Vaux [20], est donc une notion logique qui a du sens en sémantique. Mais quand les modèles catégoriques de la logique linéaire différentielle sont largement étudiés dans la littérature [18] [6] [2], peu d'axiomatiques catégoriques existent pour la logique linéaire polarisée. C'est l'objet de ce papier. Nous croyons à l'intérêt d'une axiomatique simple et élégante permettant de détecter des nouveaux principes calculatoires.

Dans ce travail, nous donnons un cadre catégorique pour la logique linéaire différentielle *finitaire* DiLL_0. Ce travail se base sur la notion de chiralité introduite par Melliès [17]. Les chiralités sont un modèle de la partie de la logique linéaire polarisée qui ne traite pas de l'ordre supérieur (la logique linéaire multiplicative). Nous étudions l'interaction entre les chiralités et les règles exponentielles de la logique linéaire différentielle finitaire : nous expliquons qu'il s'agit là d'introduire un tout petit peu d'ordre supérieur, sans néanmoins se préoccuper de la règle de la chaîne. La difficulté provient du fait que ce qui se faisait dans une seule catégorie se fait maintenant dans deux catégories mises en relation par deux négations différentes.

Organisation de l'article. Nous rappelons en partie 2 les règles de la logique linéaire différentielle, en expliquant le rôle particulier joué par la promotion. Ensuite, nous exposons les principales stuctures catégoriques permettant l'interprétation de DiLL, en discutant l'intérêt d'une axiomatique catégorique d'un calcul (partie 3). En partie 4, nous donnons la définition de notre modèle et l'interprétation des preuves et des formules. Enfin, en partie 4.4 nous démontrons l'invariance par élimination des coupures de cette sémantique.

Conventions Cette article se place à l'intersection de la correspondance de Curry-Howard et de la sémantique dénotationelle. Une fonction est un programme, et les deux sont typés par une formule. La composition de deux fonctions correspond à l'application d'une règle de coupure entre le type de ces deux fonctions. Dans le cadre de la sémantique dénotationnelle, nous ne nous intéressons pas au calcul effectif de cette fonction. Notre travail consiste justement à trouver des interprétations de la logique *invariantes par élimination des coupures*.

2 La logique linéaire différentielle

2.1 Une logique classique ou polarisée

La logique linéaire est un calcul des séquent où les règles structurelles (contraction et affaiblissement) sont restreintes à un type particulier de formules : les formules exponentielles. Ainsi, en sémantique dénotationnelle, la preuve d'un séquent $A \vdash B$ sera interprétée par une fonction linéaire $f \in \mathcal{L}(\llbracket A \rrbracket, \llbracket B \rrbracket)$. La preuve d'un séquent $!A \vdash B$ sera interprétée par un autre type de fonction, répondant à des caractéristiques plus générales $f \in \mathcal{C}(\llbracket A \rrbracket, \llbracket B \rrbracket)$. La présence de cette exponentielle démultiplie les connecteurs logiques : on retrouve en logique linéaire deux conjonctions $\otimes$ et $\&$ (multiplicatives et additives) ainsi que deux disjonctions $\parr$ et $\oplus$ (multiplicatives et additives).

La logique linéaire est d'abord (au sens historique) classique au sens où toute formule est équivalente à sa double négation linéaire : $A \equiv A^{\perp\perp}$. Cette négation transforme naturellement la conjonction additive (resp. multiplicative) en la disjonction additive (resp. multiplicative). Nous ne nous intéresserons pas ici à la logique linéaire intuitionniste, et nous représentons la plupart du temps les séquents sous-forme monolatère : le séquent $A \vdash B$ sera écrit $\vdash A^{\perp}, B$. Pour des raisons pédagogique nous pourrons toutefois les écrire sous forme bilatère.

La logique linéaire polarisée ($\mathrm{LL_P}$) [15] distingue deux types de formules suivant leur comportement logique. Sans rentrer dans les détails, nous rappelons que cette distinction a du sens en recherche de preuve [1]. Un séquent monolatère prouvable de la logique linéaire polarisée comporte *au plus* une formule positive.

Notation 1. *Les notations N, M ou P, Q désignent des formules négatives ou positives respectivement. Les notations $\mathcal{N}, \mathcal{M}$ désignent des* multi-ensembles *de formules toutes négatives. Les notations $\mathscr{N}$ et $\mathscr{P}$ désignent les catégories*

Formules négatives : $N, M := \mathfrak{n} \mid ?P \mid \uparrow P \mid N \parr M \mid \bot \mid N \,\&\, M \mid \top$.
Formules positives : $P, Q := a^\perp \mid !N \mid \downarrow N \mid P \otimes Q \mid 0 \mid P \oplus Q \mid 1$.

FIGURE 1 – Les règles de la logique linéaire polarisée $\mathrm{LL_P}$

$$\frac{}{\vdash v : A, v^\perp : A^\perp}\ \text{(axiom)} \qquad \frac{\vdash \mathcal{N}, A \qquad \vdash A^\perp, \mathcal{M}}{\vdash \mathcal{N}, \mathcal{M}}\ \text{(cut)}$$

$$\frac{}{\vdash 1}\ (1) \qquad \frac{\vdash \mathcal{N}}{\vdash \mathcal{N}, \bot}\ (\bot)$$

$$\frac{\vdash \mathcal{N}, N, M}{\vdash \mathcal{N}, N \parr M}\ (\parr) \qquad \frac{\vdash \mathcal{N}, P \qquad \vdash \mathcal{M}, Q}{\vdash \mathcal{N}, \mathcal{M}, P \otimes Q}\ (\otimes)$$

$$\frac{}{\vdash \mathcal{N}, \top}\ \top \qquad \frac{\vdash \mathcal{N}, N \qquad \vdash \mathcal{N}, M}{\vdash \mathcal{N}, N \,\&\, M}\ \&$$

$$\frac{\vdash P, \mathcal{N}}{\vdash P \oplus Q, \mathcal{N}}\ \oplus_L \qquad \frac{\vdash \mathcal{N}, Q}{\vdash \mathcal{N}, P \oplus Q}\ \oplus_R$$

FIGURE 2 – Les règles de la logique linéaire polarisée multiplicative et additive

dans lesquelles nous interprétons les formules négatives et positives respectivement.

Nous donnons en figure 1 la grammaire de la logique linéaire polarisée, construite à partir d'un ensemble de formules atomique toutes négatives $\mathfrak{n} \in \mathfrak{N}$, et en figure 2 ses règles logiques multiplicatives et additives (MALLP). La négation d'une formule positive est toujours négative et vice-versa. La négation d'une formule est définie en figure 3.

Notation 2. *Nous notons $A \multimap B := A^\perp \parr B$ l'implication linéaire entre deux formules. D'après les polarités choisies, on a donc que $P \multimap N$ est une formule négative, et cela correspond à la sémantique donnée en section 2.3.*

2.2 La logique linéaire différentielle

La logique linéaire se comprend souvent en terme de *ressources*. Une preuve de $A, !B \vdash C$ pourra utiliser plusieurs fois la formule $!B$ mais utilisera une seule fois la formule A. Une preuve interprétée par une fonction linéaire utilisera une et une seule fois son hypothèse - le domaine de la fonction linéaire.

$$(!A)^\perp = ?A^\perp \qquad (A \,\&\, B)^\perp = A^\perp \oplus B^\perp \qquad (A \oplus B)^\perp = A^\perp \,\&\, B^\perp$$
$$(?A)^\perp = !A^\perp \qquad (A \parr B)^\perp = A^\perp \otimes B^\perp \qquad (A \otimes B)^\perp = A^\perp \parr B^\perp$$
$$(\uparrow A)^\perp = \downarrow A^\perp \qquad (\downarrow A)^\perp = \uparrow A^\perp \qquad 0^\perp = \top$$
$$1^\perp = \bot \qquad \bot^\perp = 1 \qquad \top^\perp = 0$$
$$a^{\perp\perp} = a$$

FIGURE 3 – La négation linéaire involutive en logique linéaire classique

La preuve d'un séquent linéaire est en particulier non-linéaire et la logique linéaire comporte une règle permettant d'oublier que notre preuve est linéaire. C'est la règle de *déréliction* :

$$\frac{A \vdash B}{!A \vdash B}\ d$$

La logique linéaire différentielle se détache de cette intuition de ressource en permettant de *linéariser* les preuves. Ainsi, elle introduit une règle de co-déréliction :

$$\frac{!A \vdash B}{A \vdash B}\ \bar{d}$$

Rappelons que la différentielle $D_x f$ d'une fonction $E \longrightarrow F$ en un point $x \in E$ est une fonction linéaire de E dans f telle que, pour tout $h \in E$, $f(x+h) = f(x) + D_x f(h) + o(h)$.

En sémantique dénotationelle, la règle $\bar{d}$ est traditionnellement interprétée exactement par la différentiation en 0 d'une fonction. Si la preuve d'un séquent $!A \vdash B$ est interprétée par une certaine fonction différentiable $f \in \mathcal{C}^1(\llbracket A \rrbracket, \llbracket B \rrbracket)$, alors la preuve de $A \vdash B$ construite en appliquant la règle de co-déréliction au séquent précédent est interprétée par la différentielle en 0 de f :

$$D_0(f) \in \mathcal{L}(\llbracket A \rrbracket, \llbracket B \rrbracket).$$

De manière surprenante, la calcul d'une différentielle, et en particulier le calcul de la différentielle d'une composition de fonctions se fait à travers des règles tout à fait symétrique à la contraction et à l'affaiblissement. Nous rappelons d'abord les règles exponentielles de la logique linéaire sous forme bilatère.

$$\frac{\vdash \mathcal{N}}{!N \vdash \mathcal{N}}\ w \qquad \frac{!N, !N \vdash \mathcal{N}}{!N \vdash \mathcal{N}}\ c \qquad \frac{N \vdash \mathcal{N}}{!N \vdash \mathcal{N}}\ d \qquad \frac{!\mathcal{N} \vdash N}{!\mathcal{N} \vdash !N}\ p$$

Nous expliquons informellement l'interprétation logique et dénotationnelle *en termes fonctionnels* de ces règles. La règle d'affaiblissement permet au niveau logique d'introduire des hypothèses qui ne seront pas utilisées. Elle s'interprète comme la possibilité de considérer une fonction constante en une valeur $v : \llbracket \mathcal{N} \rrbracket$ comme un fonction non-linéaire sur n'importe quel domaine de définition. La règle de contraction c qui permet au niveau logique d'utiliser plusieurs fois des hypothèses exponentielles, s'interprète au niveau fonctionnel comme la multiplication scalaire entre deux fonctions. Les intuitions de la règle de déréliction ont déjà été expliquées. La règle de promotion s'interprète comme la possibilité logique de prouver plusieurs fois une formule si l'on peut se servir plusieurs fois de ses hypothèses. Au niveau fonctionnel, la règle de promotion permet de *composer des fonctions non-linéaires.* Nous expliquons maintenant pourquoi.

L'interprétation des preuves usuelles non-linéaires de la logique (minimale par exemple) se traduit au niveau fonctionnel par une adjonction entre une catégorie de fonctions non-linéaires et une catégorie de fonctions linéaires :

$$\mathcal{C}(\llbracket A \rrbracket, \llbracket B \rrbracket) \simeq \mathcal{L}(!\llbracket A \rrbracket, \llbracket B \rrbracket).$$

Ainsi, l'interprétation de la règle de promotion permet de transformer une fonction $f \in \mathcal{L}(\llbracket !A \rrbracket, \llbracket B \rrbracket)$ en une fonction $p(f) \in \mathcal{L}(\llbracket !A \rrbracket, \llbracket !B \rrbracket)$, et de composer l'interprétation de deux preuves non-linéaires $f \in \mathcal{L}(\llbracket !A \rrbracket, \llbracket B \rrbracket)$ et $g \in \mathcal{L}(\llbracket !B \rrbracket, \llbracket C \rrbracket)$ comme deux fonctions $f \in \mathcal{C}(\llbracket A \rrbracket, \llbracket B \rrbracket)$ et $g \in \mathcal{C}(\llbracket A \rrbracket, \llbracket B \rrbracket)$.

Cette adjonction entre fonctions linéaires et fonctions non-linéaires permet aussi d'interpréter, quand les fonctions non-linéaires sont des fonctions lisses, *l'exponentielle comme une espace de distributions à support compact*. Dans un modèle dénotationnel de la logique linéaire classique, l'interprétation de la négation doit être involutive. Dans une catégorie de $\mathbb{R}$-espaces vectoriels [2], cette négation linéaire est toujours interprétée comme le dual de l'interprétation de la formule : l'espace des formes linéaires, éventuellement continues, de l'interprétation de la formule. Ainsi :

$$\begin{aligned} \llbracket !A \rrbracket &\simeq \llbracket !A \rrbracket'' \\ &:= \mathcal{L}(\llbracket !A \rrbracket, \mathbb{R})' \\ &\simeq \mathcal{C}^\infty(\llbracket A \rrbracket, \mathbb{R})' \end{aligned}$$

2. Nous nous plaçons pour faire simple dans une catégorie de $\mathbb{R}$-espaces vectoriels, mais des $\mathbb{C}$-espaces vectoriels auraient pu être considérés, voir [12]

$$\frac{\vdash \mathcal{N}}{!N \vdash \mathcal{N}}\, w \qquad \frac{!N, !N \vdash \mathcal{N}}{!N \vdash \mathcal{N}}\, c \qquad \frac{N \vdash \mathcal{N}}{!N \vdash \mathcal{N}}\, d$$

$$\frac{}{\vdash !N}\, \bar{w} \qquad \frac{\vdash \mathcal{N}, !N \qquad \vdash \mathcal{M}, !N}{\vdash \mathcal{N}, \mathcal{M}, !N}\, \bar{c} \qquad \frac{\vdash \mathcal{N}, N}{\vdash \mathcal{N}, !N}\, \bar{d}$$

FIGURE 4 – Les règles exponentielles logique linéaire différentielle finitaire et polarisée ($\mathrm{DiLL}_{0,\mathrm{pol}}$)

L'ensemble des formes linéaires sur un espace de fonctions lisses $\mathcal{C}^\infty([\![A]\!], \mathbb{R})'$ est exactement l'espace des distributions à support compact $\mathscr{E}'([\![A]\!])$. Nous donnons maintenant les règles de la logique linéaire différentielle ainsi que leur interprétation en terme de distribution, pour la lectrice familière avec cette théorie. Nous renvoyons à l'article originel de Ehrhard et Regnier pour une explication en terme de fonctions [3]. Les règles exponentielles ajoutées par la logique linéaire différentielle sont les duales des règles w, c et d :

$$\frac{}{\vdash \delta_0 : !N}\, \bar{w} \qquad \frac{\vdash \mathcal{N}, \phi : !N \qquad \vdash \mathcal{M}, \psi : !N}{\vdash \mathcal{N}, \mathcal{M}, \phi * \psi : !N}\, \bar{c} \qquad \frac{\vdash \mathcal{N}, v : N}{\vdash \mathcal{N}, (f \mapsto D_0 f(v)) : !N}\, \bar{d}$$

Une distribution est un opérateur linéaire qui agit sur une fonction. Ainsi, les règles $\bar{w}$, $\bar{c}$ et $\bar{d}$ décrivent les transformations linéaires que l'on peut effectuer sur une fonction lisse. La règle de co-affaiblissement $\bar{w}$ permet d'habiter n'importe quel exponentielle par une distribution essentielle : le dirac en 0 $\delta_0; f \mapsto f(0)$. La règle de co-contraction $\bar{c}$ permet de faire le produit de convolution entre deux distributions ϕ et ψ. Surtout, elle permet de translater les distributions : pour tout opérateur de Dirac δ_x, distribution ϕ et fonction f, on a $\delta_x * \phi(f) = \phi(y \mapsto f(x+y))$. Ainsi, la règle de co-contraction convolué avec la règle de co-dereliction permet de différencier une preuve en tout point de son domaine. La règle de co-déréliction dit exactement qu'à partir n'importe quel point d'un domaine de fonction, on peut créer la distribution qui agit sur une fonction en calculant sa dérivée en 0 suivant le vecteur v.

D'après les explications données ci-dessus, la lectrice aperçoit peut être déjà qu'éliminer une coupure entre une règle de promotion et une règle de co-déréliction va demander de détailler le calcul de la règle de la chaîne propre à la différentiation des composées de fonction.

$$D_0(f \circ g)(v) = D_{f(v)} g(D_0 f(v)).$$

Le calcul de cette règle fait intervenir les trois règles co-structurelles $\bar{w}$, $\bar{c}$ and $\bar{d}$.

La logique linéaire différentielle finitaire L'introduction de DiLL permet donc une approche complètement symmétrique des règles exponentielles, exception faite de la règle de promotion. Historiquement, DiLL a d'ailleurs d'abord été introduite sous sa forme *finitaire*, c'est à dire sans la règle de promotion [3]. C'est pour cette logique là, dont les règles sont rappellées en figure 4, que nous construisons un modèle catégorique aujourd'hui. C'est une logique dont les modèles ne nécessitent pas d'ordre supérieur, et qui s'interprète donc plus facilement dans l'analyse réelle usuelle. DiLL finitaire traduit une idée d'interaction entre processus concurrents [7] et se généralise au calcul de solutions d'un opérateur différentiel partiel linéaire à coefficients constants [13]. Nous considérons notre travail sur la logique linéaire différentielle finitaire et polarisée comme une première étape vers une sémantique décomposant la règle de la chaîne via les règles co-structurelles primitives de la logique linéaire différentielle.

Définition 3. La grammaire de la logique linéaire différentielle finitaire et polarisée ($\mathrm{DiLL}_{0,P}$) est celle donnée en figure 2. Les règles de construction des preuves de la logique linéaire différentielle finitaire et polarisée consiste en l'union des règles de MALL_P (figure 2) et de celles données en figure 4.

2.3 Un premier modèle polarisé de DiLL_0

Nous donnons dans cette section un exemple de modèle polarisé de DiLL_0 en analyse réelle motivant la recherche d'un cadre catégorique. La logique linéaire différentielle provient de l'étude des modèles vectoriels de la logique

linéaire. Ces modèles sont toutefois dans un cadre restreint : il s'agit de certains espaces de suites dans lesquels les opérations se font sur les indices des suites [4], ou sur des espaces vectoriels sur un corps discret [5]. L'interprétation de DiLL dans des espaces plus généraux, comme les espaces vectoriels normés se révèle ardue, notamment lorsqu'il s'agit de combiner les contraintes d'espaces *complets* (pour une meilleure manipulation des limites et donc des fonctions différentiables) et d'espaces *reflexifs*, c'est à dire linéairement homéomorphes à leur double dual. Ainsi, *avec la contrainte d'être complet, tout modèle intuitionniste de* DiLL *ne donne pas un modèle polarisé*. En effet, l'interprétation topologiques de la construction de Chu [**?**], qui c'est à dire le fait de munir l'espace vectoriel par exemple d'une topologique faible ou d'une topologie de Mackey, ne préserve pratiquement jamais la complétion d'un espace.

Les espaces de Fréchet sont les espaces métrisables et complets. Un exemple typique d'espace de Fréchet qui n'est pas un Banach est l'espace des fonctions lisses

$$\mathcal{C}^\infty(\mathbb{R}^n, \mathbb{R})$$

muni de la famille dénombrable de semi-normes suivante :

$$p_{k,n}(f) := \sup_{|\alpha|=k} \sup_{|x|\leqslant n} \|\partial^\alpha f(x)\| \quad (n, k \in \mathbb{N}).$$

Les autres espaces de fonctions $\mathcal{C}^m$, ou les espaces de fonctions à décroissance rapide, sont également des espaces de Fréchet.

Définition 4. Pour E un espace vectoriel topologique localement convexe et séparé (lcs), on note E'_β l'espace vectoriel de toutes les formes linéaires continues sur E, muni de la topologie de convergence uniforme sur les bornés. C'est de nouveau un lcs. Un lcs E est alors dit *réflexif* lorsqu'il est linéairement homéomorphe à $(E'_\beta)'_\beta$.

Les espaces réflexifs sont naturellement ceux qui modélisent la négation linéaire involutive de DiLL. Tous les espaces de Fréchet ne sont pas réflexifs, mais tous les espaces de Fréchet *nucléaires*[3] le sont. Ceux-ci étant en plus stables par produit tensoriel et produit cartésien, on pourrait penser qu'ils forment un modèle de MALL. Pourtant, les espaces métrisables, et donc les espaces de Fréchet ne sont *pas stables par dualité*.

Les espaces métrisables sont caractérisés par leur *base dénombrable de voisinages de* 0. Le dual fort E'_β transforme cette base dénombrable d'ouverts en une *base dénombrable de bornés*. La classe des duaux d'espace de Fréchet est par contre bien connue : il s'agit des espaces DF.

Nous avons ainsi un modèle de MALL *polarisé* : les positifs y sont interprétés par des espaces nucléaires DF (une catégorie notée NDF), et les négatifs par des espaces nucléaires Fréchet (notés NF). Les différents propriétés de stabilités se trouve dans les premiers travaux sur les produits tensoriels topologiques par Grothendieck [10], dont celle liée au produit tensoriel topologique projectif complété $\tilde{\otimes}_\pi$.

$$(\mathrm{NDF}, \tilde{\otimes}_\pi, \mathbb{R}) \quad \underset{(-)'_\beta}{\overset{(-)'_\beta}{\rightleftarrows}} \quad (\mathrm{NF}^{op}, \tilde{\otimes}_\pi, \mathbb{R}) \qquad \perp \tag{1}$$

Ce modèle permet également de d'interpréter les exponentielles sur les espaces euclidiens :

$$!\mathbb{R}^n := \mathcal{C}^\infty(\mathbb{R}^n, \mathbb{R})' \text{ et } ?\mathbb{R}^n := \mathcal{C}^\infty(\mathbb{R}^n, \mathbb{R}).$$

Une extension de ces formules donne un modèle de $\mathrm{DiLL}_{0,P}$ [14]. Le travail exposé ici vise à leur donner un cadre catégorique. On fait alors face à plusieurs difficultés. Quand dans le modèle ci-dessus toutes les fonctions linéaires continues sont des morphismes de la catégorie des espaces vectoriels topologiques, il s'agit maintenant de décider quelles fonctions font partie de l'interprétation des positifs et quelles fonctions font partie de l'interprétation des négatifs. En particulier, l'interprétation de l'exponentielle fait intervenir des shifts (interprétés par une procédure de complétion) qui n'apparaissent pas lorsqu'on regarde les espaces de fonctions lisses sur $\mathbb{R}^n$, ceux ci étant déjà complets.

3. les espaces nucléairess sont les lcs dans lesquels plusieurs notions de produit tensoriel topologique correspondent. Le lecteur intéressé pourra se référer par exemple à [11]

2.4 Elimination des coupures

Nous rappelons dans cette section les règles d'élimination des coupures pour les règles exponentielles de DiLL0. Ces règles correspondent aux intuitions sémantiques de l'analyse réelle : ainsi, l'élimination des coupures entre c et $\bar{d}$ correspond à la différentiation d'une multiplication scalaire de fonctions, donc à la formule $(fg)' = f'g + g'f$. Nous renvoyons à [3] pour une explication des règles en ces termes.

Comme indiqué par l'exemple de la règle de Leibneiz ci-dessus, les preuves de la logique linéaire différentielle doivent être sommées. Elle obéissent donc à une grammaire particulière. Les preuves de DiLL sont des sommes commutatives et assocatives d'arbres de preuve. Il y a pour cette somme un élément neutre : toute formule A possède une 0-preuve, notée $\mathfrak{o}_A$.

$$\dfrac{\dfrac{\vdash \Gamma}{\vdash \Gamma, ?E}\, w \qquad \dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w}}{\vdash \Gamma}\,\text{cut} \rightsquigarrow \vdash \Gamma$$

$$\dfrac{\dfrac{\vdash \Gamma, ?E, ?E}{\vdash \Gamma, ?E}\, c \qquad \dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w}}{\vdash \Gamma}\,\text{cut} \rightsquigarrow \dfrac{\dfrac{\vdash \Gamma, ?E, ?E \qquad \dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w}}{\vdash \Gamma, ?E}\,\text{cut} \qquad \dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w}}{\vdash \Gamma}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash \Gamma, !E^\perp \qquad \vdash \Gamma', !E^\perp}{\vdash \Gamma, \Gamma', !E^\perp}\, \bar{c} \qquad \dfrac{\vdash \Delta}{\vdash \Delta, ?E}\, w}{\vdash \Gamma, \Gamma', \Delta}\,\text{cut} \rightsquigarrow \dfrac{\vdash \Gamma, !E^\perp \qquad \dfrac{\vdash \Delta}{\vdash \Delta, ?E}\, w}{\dfrac{\vdash \Gamma, \Delta}{\vdash \Gamma, \Delta, ?E}\, w \qquad \vdash \Gamma', !E^\perp}\,\text{cut}}{\vdash \Gamma, \Gamma', \Delta}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash \Delta, E}{\vdash \Delta, ?E}\, d \qquad \dfrac{\vdash \Gamma, E^\perp}{\vdash \Gamma, !E^\perp}\, \bar{d}}{\vdash \Gamma, \Delta}\,\text{cut} \rightsquigarrow \dfrac{\vdash \Delta, E \qquad \vdash \Gamma, E^\perp}{\vdash \Gamma, \Delta}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash \Gamma}{\vdash \Gamma, ?E}\, w \qquad \dfrac{\vdash \Delta, E^\perp}{\vdash \Delta, !E^\perp}\, \bar{d}}{\vdash \Gamma, \Delta}\,\text{cut} \rightsquigarrow \mathfrak{o}_{\Gamma \parr \Delta}$$

$$\dfrac{\dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w} \qquad \dfrac{\vdash \Gamma, E}{\vdash \Gamma, ?E}\, d}{\vdash \Gamma}\,\text{cut} \rightsquigarrow \mathfrak{o}_{\Gamma \parr \Delta}$$

$$\dfrac{\dfrac{\vdash \Gamma, ?E, ?E}{\vdash \Gamma, ?E}\, c \qquad \dfrac{\vdash \Delta, E^\perp}{\vdash \Delta, !E^\perp}\, \bar{d}}{\vdash \Gamma, \Delta}\,\text{cut} \rightsquigarrow$$

$$\dfrac{\dfrac{\vdash \Gamma, ?E, ?E \qquad \dfrac{\vdash \Delta, E^\perp}{\vdash \Delta, !E^\perp}\, \bar{d}}{\vdash \Gamma, \Delta, ?E}\,\text{cut} \qquad \dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w}}{\vdash \Delta, \Gamma}\,\text{cut} \;+\; \dfrac{\dfrac{\vdash \Gamma, ?E, ?E \qquad \dfrac{\vdash}{\vdash\, !E^\perp}\, \bar{w}}{\vdash \Gamma, \Delta, ?E}\,\text{cut} \qquad \dfrac{\vdash \Delta, E^\perp}{\vdash \Delta, !E^\perp}\, \bar{d}}{\vdash \Delta, \Gamma}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash \Gamma, !E^\perp \qquad \vdash \Gamma', !E^\perp}{\vdash \Gamma, \Gamma', !E^\perp}\,\bar{c} \qquad \dfrac{\vdash \Delta, E}{\vdash \Delta, ?E}\,d}{\vdash \Gamma, \Gamma', \Delta}\,\text{cut} \rightsquigarrow$$

$$\dfrac{\dfrac{\vdash \Gamma, !E^\perp \qquad \dfrac{\vdash \Delta, E}{\vdash \Delta, ?E}\,d}{\dfrac{\vdash \Gamma, \Delta}{\vdash \Gamma, \Delta, ?E}\,w} \qquad \vdash \Gamma', !E^\perp}{\vdash \Gamma', \Gamma, \Delta}\,\text{cut} \quad + \quad \dfrac{\vdash \Gamma', !E^\perp \qquad \dfrac{\vdash \Delta, E}{\vdash \Delta, ?E}\,d}{\dfrac{\vdash \Gamma', \Delta}{\vdash \Gamma'.\Delta, ?E}\,w} \qquad \vdash \Gamma, !E^\perp}{\vdash \Gamma', \Gamma, \Delta}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash \Delta, ?E, ?E}{\vdash ?E}\,c \qquad \dfrac{\vdash \Gamma, \phi : !E^\perp \qquad \vdash \Gamma', \psi : !E^\perp}{\vdash \Gamma, \Gamma', !E^\perp}\,\bar{c}}{\vdash \Gamma, \Gamma', \Delta}\,\text{cut} \rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ?E, !E^\perp}\,(\text{ax}) \qquad \dfrac{}{\vdash ?E, !E^\perp}\,(\text{ax})}{\vdash ?E, ?E, !E^\perp}\,\bar{c} \qquad \vdash \Delta, ?E, ?E}{\vdash \Delta, ?E, ?E, ?E}\,cut \qquad \dfrac{\dfrac{}{\vdash ?E, !E^\perp}\,(\text{ax}) \qquad \dfrac{}{\vdash ?E, !E^\perp}\,(\text{ax})}{\vdash ?E, ?E, !E^\perp}\,\bar{c}}{\vdash \Delta, ?E, ?E, ?E, ?E}\,\text{cut}$$

$$\vdots$$
$$\pi$$
$$\vdots$$

où π est la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash \Delta, ?E, ?E, ?E, ?E}\,c}{\vdash \Delta, ?E, ?E, ?E} \qquad \vdash \Gamma, !E^\perp}{\dfrac{\vdash \Delta, \Gamma, ?E, ?E}{\vdash \Delta, \Gamma, ?E}\,c \qquad \vdash \Gamma', !E^\perp}\,\text{cut}}{\vdash \Delta, \Gamma, \Gamma'}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash \Gamma, E}{\vdash \Gamma, \uparrow E}\,(\uparrow) \qquad \dfrac{\vdash \Delta, E^\perp}{\vdash \Delta, \downarrow E^\perp}\,(\downarrow)}{\vdash \Gamma, \Delta}\,\text{cut} \rightsquigarrow \dfrac{\vdash \Gamma, E \qquad \vdash \Delta, E^\perp}{\vdash \Gamma, \Delta}\,\text{cut}$$

3 Quelques notions catégoriques

L'apparition de la logique linéaire différentielle a engendré l'étude de diverses catégories différentielles [2]. Ces structures axiomatiques sont des généralisations de modèles de DiLL, et cherchent à se rapprocher de la géométrie en s'éloignant déjà de l'interprétation des principes calculatoires en jeu. Un premier travail de Fiore [18] met en évidence le rôle important des connecteurs additifs pour l'interprétation des règles exponentielles : quand multiplication et conjonction additives correspondent, alors l'interprétation de w , c, $\bar{w}$, $\bar{c}$ découle de la monoïdalité forte de l'interprétation de !. Mais le travail s'attache à interpréter la logique linéaire différentielle *intuitionniste*. Ici, nous voulons mettre en évidence dans les catégories l'importance de la négation linéaire involutive dans la syntaxe de DiLL et en analyse. Dans une autre étape de recherche, nous souhaiterions tirer profit des structures polarisées pour mieux interpréter les règles co-structurelles de la logique linéaire différentielle (voir section 5).

Remarque 5. Nous supposons connu le langage de base des catégories - foncteurs, transformation naturelle, (co)-monade, adjonctions. Nous rappelons ici quelques définitions fondamentales à l'interprétation des règles de LL.

Notation 6. *Soit $(\mathcal{C}, \otimes, 1)$ une catégorie monoïdale symétrique, on adoptera les notations suivantes pour les isomorphismes naturels accompagnant cette structure :*

— *morphismes d'associativité :* $\alpha^{\otimes}_{A,B,C} : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C)$.

— *morphismes de neutralité :* $\rho^{\otimes}_{A} : A \otimes 1 \simeq A, \lambda^{\otimes}_{A} : 1 \otimes A \simeq A$.

— *morphisme de commutativité :* $\gamma^{\otimes}_{A,B} : A \otimes B \simeq B \otimes A$.

Sans ambiguïté on notera respectivement $\alpha_{A,B,C}, \rho_A, \lambda_A, \gamma_{A,B}$ voire $\alpha, \rho, \lambda, \gamma$ pour $\alpha^{\otimes}_{A,B,C}, \rho^{\otimes}_{A}, \lambda^{\otimes}_{A}, \gamma^{\otimes}_{A,B}$.

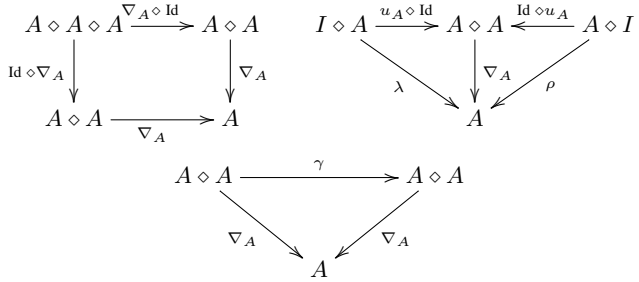
3.1 Biproduit et règles costructurelles

Nous décrivons dans cette partie le rôle joué par le biproduit dans les modèles non-polarisés de la logique linéaire différentielle, d'après les travaux de Fiore.

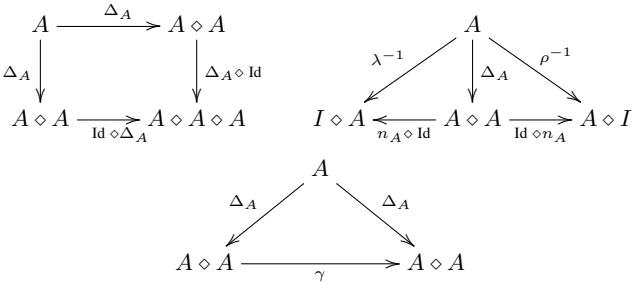
Notation 7. *Soit F un foncteur monoïdal fort de $(\mathcal{C}, \diamond, I)$ vers $(\mathcal{D}, \blacklozenge, J)$, on note $m^{A,B}_{F,\diamond,\blacklozenge}$ le morphisme de $F(A \diamond B)$ vers $FA \blacklozenge FB$ et on notera $m^{0}_{F,\diamond,\blacklozenge}$ le morphisme de FI vers J.*

On notera $m^{A,B}_{F}, m_F$ voire m quand il n'y a pas d'ambiguïté.

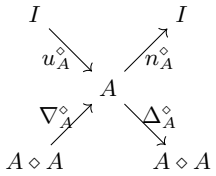
Définition 8. Soient $\mathcal{L}$ une catégorie, $\diamond$ une loi monoïdale symétrique sur $\mathcal{L}$ et soit A un objet de $\mathcal{L}$, avec $u_A : I \longrightarrow A$ et $\nabla_A : A \diamond A \longrightarrow A$ des morphismes. On dit que (A, u_A, ∇_A) est un *monoïde commutatif* si les diagrammes suivants commutent :



Soient des morphismes $n_A : A \longrightarrow I$ et $\Delta_A : A \longrightarrow A \diamond A$, (A, n_A, Δ_A) est un *co-monoïde commutatif* si les diagrammes suivants commutent :



Définition 9. Un *biproduit* sur une catégorie $\mathcal{L}$ est une structure monoïdale $(\diamond, I)$ avec des transformations naturelles :



telles que $(A, u_A^\diamond, \nabla_A^\diamond)$ est un monoïde commutatif et $(A, n_A^\diamond, \Delta_A^\diamond)$ est un co-monoïde commutatif.
On omettra les $\diamond$ au-dessus des morphismes quand il n'y aura pas d'ambiguïté.

Exemple 10. L'exemple typique d'un biproduit est le produit des espaces vectoriels : pour tout espaces vectoriels E et F, $E \times F \simeq E \oplus F$.

Proposition 11. *Le biproduit est un produit avec projections :* $n \diamond Id; \lambda^\diamond$ *et* $Id \diamond n; \rho^\diamond$. *C'est aussi un co-produit avec les co-projections* $(\rho^\diamond)^{-1}$; $Id \diamond u$ *et* $(\lambda^\diamond)^{-1}; u \diamond Id$.

Remarque 12. Le biproduit sur une catégorie est ce qui permet *d'additionner les morphismes* et donc d'interpréter les sommes de preuves de DiLL. En effet, la présence d'une structure de biproduit est équivalente à la conjonction d'un enrichissement sur la catégorie des monoïdes et à la présence d'un produit [18, 2.3]. Ainsi, pour deux morphismes $f, g \in \mathcal{L}(A, B)$, on construit leur somme :

$$f + g : A \xrightarrow{\Delta} A \diamond A \xrightarrow{f \diamond g} B \diamond B \xrightarrow{\nabla} B.$$

Nous rappelons par ailleurs que tout produit sur une catégorie donne lieu à un morphisme diagonal :

Proposition 13. *[16, Prop 16] Soit* $(\mathcal{C}, \times, 0)$ *une catégorie cartésienne, avec* $n_A : A \longrightarrow 0$ *comme morphisme terminal. Alors* $\mathcal{C}$ *est muni de la transformation naturelle suivante :*

$$\Delta_A : A \longrightarrow A \times A$$

telle que pour tout objet A *(*A*,* Δ_A*,* n_A*) est un co-monoïde commutatif.*

De même, toute catégorie co-cartésienne $(\mathcal{C}, \otimes, 0)$ est naturellement munie d'une co-diagonale $\nabla_A : A \oplus A \longrightarrow A$.

3.2 Chiralités

Les chiralités ont été introduites par Melliès [17] comme une manière de relâcher l'injonction d'une négation linéaire involutive en une équivalence de catégories. Dans leur première forme, les chiralités consistent en deux foncteurs contravariants formant une équivalence entre deux catégories monoïdales, accompagnés de foncteurs covariants permettant le plongement des catégories l'une dans l'autre. Ces deux paires de foncteurs représentent respectivement les deux négations (des positifs aux négatifs et vice-versa) et les deux shifts.

Nous poussons le raisonnement plus loin en demandant que la composition des foncteurs soit un isomorphisme non pas sur les deux catégories mais sur une seule d'entre elles. Ce cas de figure est celui qui apparaît concrètement dans les modèles dénotationnels, où seules les interprétations des formules négatives sont par construction invariantes par double négation. L'interprétation des formules positives nécessite justement l'application d'un shift pour vérifier cette invariance.

Définition 14. Une *fermeture polarisée à gauche* (resp. *à droite*) est une adjonction $L : \mathscr{P} \longrightarrow \mathscr{N} \dashv R : \mathscr{N} \longrightarrow \mathscr{P}$ telle qu'on ait un isomorphisme naturel entre $\mathrm{Id}_{\mathscr{P}}$ et $R \circ L$ (resp. un isomorphisme naturel entre $L \circ R$ et $\mathrm{Id}_{\mathscr{N}}$) qui soit image de l'identité par l'adjonction.

Les chiralités négatives définies ci-dessous sont donc une structure intermédiaire entre les chiralités de dialogue et les chiralités mixtes définies par Melliès.

Définition 15. Une *chiralité négative* est la donnée d'une paire de catégories monoïdales $(\mathscr{P}, \otimes, 1)$ et $(\mathscr{N}, \parr, \bot)$ avec :

- une fermeture polarisée à gauche fortement monoïdale, $(-)^{\bot_L} : \mathscr{P} \longrightarrow \mathscr{N}^{op} \dashv (-)^{\bot_R} : \mathscr{N}^{op} \longrightarrow \mathscr{P}$,
- une fermeture polarisée à gauche, $\uparrow : \mathscr{P} \longrightarrow \mathscr{N} \dashv \downarrow : \mathscr{N} \longrightarrow \mathscr{P}$,
- une famille de bijections :

$$\chi_{p,n,m} : \mathscr{N}(\uparrow p, n \parr m) \simeq \mathscr{N}(\uparrow(p \otimes n^{\bot_R}), m) \tag{2}$$

naturelles en p, n et m telles que le diagramme suivant commute :

$$\begin{array}{ccccc} \mathscr{N}(\uparrow(p \otimes (n' \parr n)^{\perp_R}), m) & & \xrightarrow{\quad\chi\quad} & & \mathscr{N}(\uparrow p, (n' \parr n) \parr m) \\ \Big\downarrow{\scriptstyle \text{associativité, monoïdalité de la négation}} & & & & \Big\uparrow{\scriptstyle \text{associativité}} \\ \mathscr{N}(\uparrow((p \otimes n'^{\perp_R}) \otimes n^{\perp_R}), m) & \xrightarrow{\chi} & \mathscr{N}(\uparrow p \otimes n'^{\perp_R}, n \parr m) & \xrightarrow{\chi} & \mathscr{N}(\uparrow p, n' \parr (n \parr m)) \end{array} \tag{3}$$

On omettra les indices sur χ quand il n'y aura pas d'ambiguïté.

Remarque 16. Dans la suite, $\chi_n(f)$ signifiera qu'on veut faire passer n de *droite à gauche*, ainsi, si on a que f est un morphisme de $\uparrow p$ vers $m \parr n$, $\chi_n(f)$ désignera le morphisme $\chi_{p,n,m}(f;\gamma)$ et si f est un morphisme de $\uparrow p$ vers $n \parr m$, $\chi_n(f)$ désignera $\chi_{p,n,m}(f)$.

4 Un modèle catégorique polarisé de $\mathrm{DiLL}_{0,pol}$

Dans cette partie nous donnons un cadre catégorique pour interpréter les règles de la logique linéaire différentielle finitaire et polarisée, de manière invariante par élimination des coupures. Ce modèle consiste en une chiralité négative, interprétant les connecteurs multiplicatifs, un biproduit *sur chacune des catégorie de la chiralité*, ainsi qu'un opérateur de co-déréliction interprétant la règle du même nom.

Une difficulté de ce modèle provient du fait que l'interprétation d'une preuve fait intervenir les plongements covariants d'une catégorie dans une autre (les shifts $\uparrow$ et $\downarrow$) : les structures doivent donc être compatibles avec ceux-ci. De manière intéressante, exiger un biproduit sur l'une des catégorie ne semble pas suffire ; nous avons besoin d'un biproduit sur chacune des deux catégories de la chiralité.

Une deuxième difficulté provient de la structure exponentielle. Les lois régissant ! sont traditionnellement modelisées par une adjonction entre une catégorie $\mathcal{L}$ aux morphismes linéaires et une catégories $\mathcal{L}_!$ aux morphismes non-linéaires [16]. Ces deux adjonctions doivent maintenant interagir avec les deux catégories représentant la polarisation. Ainsi, l'interprétation de l'exponentielle ! et de son dual ? font explicitement intervenir des shifts $\uparrow$ et $\downarrow$ (voir définition 20).

Définition 17. Un modèle de $\mathrm{DiLL}_{0,\mathrm{pol}}$ consiste en :

$\perp$ Une chiralité négative $(\mathscr{P}, \otimes, 1)$ et $(\mathscr{N}, \parr, \perp)$ avec une clôture monoïdal forte $(-)^{\perp_L} : \mathscr{P} \longrightarrow \mathscr{N}^{op} \dashv (-)^{\perp_R} : \mathscr{N}^{op} \longrightarrow \mathscr{P}$, et une clôture polarisée $\uparrow : \mathscr{P} \longrightarrow \mathscr{N} \dashv \downarrow : \mathscr{N} \longrightarrow \mathscr{P}$. Nous noterons χ la famille de bijections : $\chi : \chi_{p,n,m} : \mathscr{N}(\uparrow p, n \parr m) \simeq \mathscr{N}(\uparrow(p \otimes n^{\perp_R}), m)$.

Notation 18. *On notera refl l'isomorphisme de $(-)^{\perp_R \perp_L}$ vers Id dans $\mathscr{N}$, et compl l'isomorphisme de $\uparrow \circ \downarrow$ vers Id dans $\mathscr{N}$.*

$\oplus$ Un *biproduit* $(\oplus, I)$ sur $\mathscr{P}$. Nous notons $I \xrightarrow{u_P^\oplus} P$, $P \oplus P \xrightarrow{\nabla_P^\oplus} P$, $P \xrightarrow{n_P^\oplus} I$ et $P \xrightarrow{\Delta_P^\oplus} P \oplus P$ les transformations naturelles telles que $(P, u^\oplus, \nabla^\oplus)$ est un monoïde commutatif et $(P, n^\oplus, \Delta^\oplus)$ est un co-monoïde commutatif.

$\&$ Un *biproduit* $(\&, J)$ sur $\mathscr{N}$. Nous notons $J \xrightarrow{u_N^\&} N$, $N \,\&\, N \xrightarrow{\nabla_N^\&} N$, $N \xrightarrow{n_N^\&} J$ et $N \xrightarrow{\Delta_N^\&} N \,\&\, N$ les transformations naturelles telles que $(N, u^\&, \nabla^\&)$ est un monoïde commutatif et $(N, n^\&, \Delta^\&)$ est un co-monoïde commutatif.

$\oplus_\infty$ Une catégorie co-cartésienne $(\mathscr{P}^\infty, \oplus_\infty, 0)$

$\otimes$ Une clôture monoïdale forte

$$\mathscr{E} : (\mathscr{P}^{\infty,op}, \oplus, 0_{\mathscr{P}^\infty}) \longrightarrow (\mathscr{N}^{op}, \parr, \perp) \dashv \mathcal{U} : (\mathscr{N}^{op}, \parr, \perp) \longrightarrow (\mathscr{P}^{\infty,op}, \oplus, 0_{\mathscr{P}^\infty}).$$

Notation 19. *On note $d : \mathscr{E} \circ \mathcal{U} \longrightarrow Id$ (ici dans $\mathscr{N}^{op}$) l'unité de l'adjonction précédente entre $\mathscr{E}$ et $\mathcal{U}$. Pour tout $N \in \mathscr{N}$, d_N est donc un morphisme de $\mathscr{N}$ qui va de N vers $\mathscr{E} \circ \mathcal{U}(N)$.*

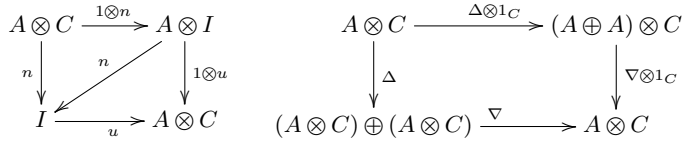
$clos_P$ Une famille d'isomorphismes naturels dans $\mathscr{P}$:

$$\mathrm{clos}_P : {\downarrow}P^{\perp_L} \simeq (\uparrow P)^{\perp_R}.$$

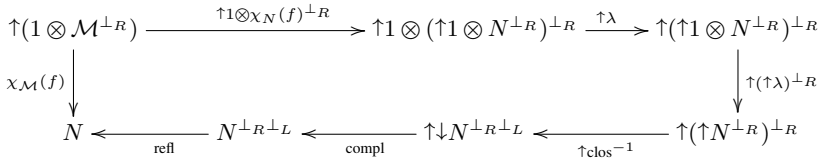
$\bar{d}$ Une transformation naturelle $\bar{d} : \mathscr{E} \circ \mathcal{U} \longrightarrow Id$ vérifiant :

$$d_N ; \bar{d}_N = \mathrm{Id}_N$$

$\oplus/\otimes$ On demande que $\oplus$ soit compatible avec $\otimes$, c'est à dire que les diagrammes suivants commutent :



$\chi/\otimes$ On demande en plus que le diagramme suivant commute pour tout $N, \mathcal{M} \in \mathscr{N}$ et tout morphisme $f \in \mathscr{N}(\uparrow 1, N \parr \mathcal{M})$:



(4)

Définition 20. On notera ? le foncteur $\mathscr{E} \circ \mathcal{U} \circ \uparrow$, ainsi que ! le foncteur $(\mathscr{E} \circ \mathcal{U} \circ \uparrow(-)^{\perp_R})^{\perp_R}$.

Remarque 21. La présence d'un produit sur $\mathscr{N}$ et d'un coproduit sur $\mathscr{P}$ est nécessaire à l'interprétation des règles multiplicatives et additives polarisées. Il nous faut aussi sommer les interprétations des preuves de DiLL. Celles-ci seront interprétées comme des morphismes dans $\mathscr{N}(\uparrow P, N)$. Suivant la remarque 12 il faudrait donc disposer d'un morphisme diagonal $\Delta_P : P \longrightarrow P \oplus P$ (modulo $\uparrow$) et d'un morphisme co-diagonal $\nabla_N : N \,\&\, N \longrightarrow N$. Ceux-ci sont peu ou prou équivalent respectivement à la présence d'un produit sur $\mathscr{P}$ et d'un coproduit sur $\mathscr{N}$ [16, prop 16]. Ainsi, dans un premier temps, nous avons demandé un biproduit sur chacune des catégories $\mathscr{P}$ et $\mathscr{N}$. Nous espérons dans un deuxième temps pouvoir alléger cette structure.

4.1 Premières propriétés de notre modèle

Soit $\mathcal{L}$ un modèle de $\mathrm{DiLL}_{\mathrm{pol}}$ comme introduit en définition 17. Avant de nous attaquer à l'interprétation des formules, nous démontrons dans cette section des propriétés dans ce modèle. Les preuves des résultats énoncés ici se trouvent dans la version longue de l'article disponible sur HAL.

Les résultats suivants proviennent du fait que les adjoints à droite préservent les limites :

Proposition 22. *Le foncteur $\uparrow$ est monoïdal fort entre les catégories $(\mathscr{P}, \oplus, I)$ et $(\mathscr{N}, \&, J)$.*

Proposition 23. *Le foncteur $\mathcal{U}$ est monoïdal fort entre $(\mathcal{N}, \&, J)$ et $(\mathcal{P}^{\infty}, \oplus_{\infty}, 0)$.*

Corollaire 24. Le foncteur ? est monoïdal fort de $(\mathcal{P}, \oplus, I)$ à $(\mathcal{N}, \parr, \bot)$.

Les morphismes suivants serviront à interpréter la contraction, l'affaiblissement, la co-contraction et le co-affaiblissement :

$$c_P := (m^{P,P}_{?,\oplus,\parr})^{-1} ; ?\nabla_P : ?P \parr ?P \longrightarrow ?P$$

$$w_P := (m^0_?)^{-1} ; ?u_P : \bot \longrightarrow ?P$$

$$\bar{c}_P := ?\Delta_P ; m^{P,P}_? : ?P \longrightarrow ?P \parr ?P$$

$$\bar{w}_P := ?n_P; m?_0 : ?P \longrightarrow \bot$$

Nous donnons la structure additive des morphismes sur $\mathscr{N}(N, M)$:

$$f + g := \Delta_N^{\&}; f \,\&\, g; \nabla_M \text{ et } 0_{(N,M)} := n_N^{\&}; u_M^{\&}.$$

4.2 Interprétation des formules

Les formules négatives sont interprétées par des objets de $\mathscr{N}$ quand les formules positives sont interprétées par des objets de $\mathscr{P}$. A l'aide d'une première interprétation des formules atomiques dans $\mathscr{N}$, l'interprétation des formules se fait par induction sur la grammaire des formules de DiLL. On utilise pour cela les foncteurs définis précédemment, avec des notations correspondant aux constructeurs logiques employés.

Proposition 25. *On interprète la négation d'une formule à l'aide des foncteurs de dualité, de manière à ce que le dual de l'interprétation d'une formule et* l'interprétation de la négation de la formule *soient isomorphes. On note ces isomorphismes* $isopos_P : [\![P^\perp]\!] \longrightarrow [\![P]\!]^{\perp_L}$ *et* $isoneg_N : [\![N^\perp]\!] \longrightarrow [\![N]\!]^{\perp_R}$ *et on les définit par induction sur les formules. En particulier, afin d'assurer la proposition 26, on s'assure d'utiliser la forte monoïdalité de* $\perp_R$ *et non celle de* $\perp_L$ *(les deux étaient possibles mais un choix était nécessaire).*

- $n^\perp$ *Si* $P = n^\perp$ *(la négation d'une variable) :* $isopos_P := refl^{-1}$.
- n *Si* $N = n$ *(une variable) :* $isoneg_N := Id_{[\![n]\!]}$.
- $\otimes$ *Si* $P = P_1 \otimes P_2$ *:* $isopos_P := refl^{-1}; ((m_{\perp_R}^{[\![P_1^\perp]\!],[\![P_2^\perp]\!]})^{-1})^{\perp_L}; (isoneg_{P_1^\perp} \otimes isoneg_{P_2^\perp})^{\perp_L}$.
- $\parr$ *Si* $N = N_1 \parr N_2$ *:* $isoneg_N := isoneg_{N_1} \otimes isoneg_{N_2}; (m_{\perp_R}^{[\![N_1]\!],[\![N_2]\!]})^{-1}$.
- $\oplus$ *Le cas* $P = P_1 \oplus P_2$ *se traite symétriquement au cas* $P_1 \otimes P_2$.
- $\&$ *Le cas* $N = N_1 \,\&\, N_2$ *se traite symétriquement au cas* $N_1 \parr N_2$.
- $!$ *Si* $P = !N$ *:* $isopos_{!N} := ?isoneg_N; refl^{-1}$.
- $?$ *Si* $N = ?P$ *:* $isoneg_{?P} := (?isoneg_{P^\perp})^{\perp_R}$.
- $\downarrow$ *Si* $P = \downarrow N$ *:* $isopos_{\downarrow N} := refl^{-1}; (clos)^{\perp_L}; (\downarrow isopos_{N^\perp})^{\perp_L}$.
- $\uparrow$ *Si* $N = \uparrow P$ *:* $isoneg_{\uparrow P} := \downarrow isopos_P; clos$.
- 1 *Si* $P = 1$ *:* $isopos_1 := refl^{-1}; ((m_{\perp_R}^0)^{-1})^{\perp_L}$.
- $\perp$ *Si* $N = \perp$ *:* $isoneg_\perp := (m_{\perp_R}^0)^{-1}$.
- $\top$,0 *Les cas* $N = \top$ *ou* $P = 0$ *se traitent symétriquement aux cas* $N = \perp$ *et* $P = 1$.

On interprète des *ensembles de formules négatives* $\mathcal{N} := N_1, ..., N_n$ par la disjonction multiplicative de l'interprétation des formules :

$$[\![\mathcal{N}]\!] := [\![N_1]\!] \parr [\![N_2]\!] \parr \cdots \parr [\![N_n]\!].$$

La proposition suivante est immédiate :

Proposition 26. *Pour toute formule négative* N*, on a :*

$$(isoneg_N)^{\perp_L}; {isopos_{N^\perp}}^{-1} = refl_{[\![N]\!]}.$$

4.3 Interprétation des preuves

Nous donnons maintenant l'interprétation des preuves de notre système. Comme dans les chiralités de dialogue, La *preuve d'un séquent* $\vdash P, \mathcal{N}$ avec $\mathcal{N}$ un multi-ensemble de formules négatives et P une formule positive est interprétée par un *morphisme dans* $\mathscr{N}([\![P]\!]^{\perp_L}, [\![\mathcal{N}]\!])$. La preuve d'un séquent $\vdash \mathcal{N}$ avec $\mathcal{N}$ un multi-ensemble de formules négatives est interprétée par un morphisme dans $\mathscr{N}(\uparrow 1, [\![\mathcal{N}]\!])$.
Cette définition est traditionnellement faite par induction sur la dernière règle utilisée dans la preuve.

Remarque 27. Un séquent est ici par définition un multi-ensemble de formules, le séquent $\vdash A, B$ n'est donc pas différent de $\vdash B, A$, cependant $[\![A]\!] \parr [\![B]\!]$ et $[\![B]\!] \parr [\![A]\!]$ sont deux objets différents dans une catégorie. Pour être rigoureux, il ne faudrait donc pas donner une interprétation des séquents, mais plutôt d'une version des séquents où l'ordre des formules du séquent à son importance. Ici on fixe un ordre arbitraire sur l'ordre de nos formules et on omet les morphismes de commutativités qu'il faudrait rajouter dans certains cas.

cut Soit $f : \uparrow 1 \longrightarrow [\![\mathcal{N}]\!] \parr [\![N]\!]$ l'interprétation d'une preuve de $\vdash \mathcal{N}, N$ et $g : [\![N^\perp]\!]^{\perp_L} \longrightarrow [\![\mathcal{M}]\!]$ l'interprétation d'une preuve de $\vdash N^\perp, \mathcal{M}$. On interprète la preuve de $\vdash [\![\mathcal{N}]\!], [\![\mathcal{M}]\!]$ résultant d'une coupure entre les deux précédentes preuves par $f; [\![\mathcal{N}]\!] \parr (\text{isopos}; g)$.

ax Pour toute formule N, l'axiome prouvant $\vdash N, N^{\perp_R}$ s'interprète par $\text{refl} : [\![N]\!]^{\perp_R \perp_L} \longrightarrow [\![N]\!]$.

$\otimes$ Soit $f : [\![P]\!]^{\perp_L} \longrightarrow [\![\mathcal{N}]\!]$ interprétant une preuve de $\vdash P, \mathcal{N}$ et $g : [\![Q]\!]^{\perp_L} \longrightarrow [\![\mathcal{M}]\!]$ interprétant une preuve de $\vdash Q, \mathcal{M}$. On définit l'interprétation de $\vdash P \otimes Q, \mathcal{N}, \mathcal{M}$ résultant de la règle $(\otimes)$ appliquée aux deux preuves précédentes par : $\text{isopos}^{-1}; \text{isopos} \parr \text{isopos}; f \parr g : [\![P \otimes Q]\!]^{\perp_L} \longrightarrow [\![\mathcal{N}]\!] \parr [\![\mathcal{M}]\!]$.

$\parr$ Soit $f \in \mathscr{M}(\uparrow 1, [\![\mathcal{N}, N_1, N_2]\!]$ interprétant une preuve du séquent $\vdash \mathcal{N}, N_1, N_2$. L'interprétation de la preuve du séquent $\vdash \mathcal{N}, N_1 \parr N_2$ est également f.

$\oplus$ Soit $f : [\![P]\!]^{\perp_L} \longrightarrow [\![\mathcal{N}]\!]$ interprétant une preuve de $\vdash P, \mathcal{N}$. On définit l'interprétation de la preuve de $\vdash P \oplus Q, \mathcal{N}$ résultant par application de la règle $(\oplus_g)$ par : $\text{isopos}^{-1}; \text{isopos} \,\&\, \text{isopos}; \text{Id} \,\&\, n_{[\![Q]\!]^{\perp_L}}; \rho; f : (P \,\&\, Q)^{\perp_L} \longrightarrow [\![\mathcal{N}]\!]$. Le cas de la règle $(\oplus_d)$ se traite de manière symétrique.

$\&$ Soit $f : \uparrow 1 \longrightarrow [\![N]\!] \parr [\![\mathcal{N}]\!]$ interprétant une preuve de $\vdash N, \mathcal{N}$ et $g : \uparrow 1 \longrightarrow [\![M]\!] \parr [\![\mathcal{N}]\!]$ interprétant une preuve de $\vdash M, \mathcal{N}$. On définit l'interprétation de la preuve de $\vdash N \,\&\, M, \mathcal{N}$ résultant par application de la règle $(\&)$ aux précédentes preuves : $\chi^{-1}_{[\![\mathcal{N}]\!]}(\Delta; \chi_{[\![\mathcal{N}]\!]}(f) \,\&\, \chi_{[\![\mathcal{N}]\!]}(g))$.

1 On interprète la preuve qui ne contient que la règle (1) par $m^0_{\perp_L, \otimes, \parr} \in \mathscr{M}(1^{\perp_L}, \perp)$.

$\perp$ Soit $f : \uparrow 1 \longrightarrow [\![\mathcal{N}]\!]$ l'interprétation d'une preuve de $\vdash \mathcal{N}$. On interprète la preuve de $\vdash \mathcal{N}, \perp$ qui résulte de l'application de la règle $(\perp)$ par $f; \rho^{-1}$.

$\top$ On interprète la preuve ne comportant que cette règle par la zéro-preuve $0_{\mathcal{N}(\uparrow 1, J \parr [\![\mathcal{N}]\!])}$.

c Soit $f : \uparrow 1 \longrightarrow [\![\mathcal{N}]\!] \parr ?[\![P]\!] \parr ?[\![P]\!]$ interprétant $\vdash \mathcal{N}, ?P, ?P$. On construit $f; \mathcal{N} \parr c_{[\![P]\!]} : \uparrow 1 \longrightarrow [\![\mathcal{N}]\!] \parr ?[\![P]\!]$ qui interprète $\vdash \mathcal{N}, ?P$.

$\bar{c}$ Pour la co-contraction, on a $f : (![\![N]\!])^{\perp_L} \longrightarrow [\![\mathcal{N}]\!]$ qui interprète $\vdash \mathcal{N}, !N$ et $g : (![\![N]\!])^{\perp_L} \longrightarrow [\![\mathcal{M}]\!]$ qui interprète $\vdash \mathcal{M}, !N$. Et donc on a $(\text{ isopos}); \bar{c}_{[\![N^\perp]\!]}; ((\text{isopos})^{-1}; f) \parr ((\text{isopos})^{-1}; g) : ?[\![N^\perp]\!] \longrightarrow [\![\mathcal{N}]\!] \parr [\![\mathcal{M}]\!]$ qui interprète $\vdash \mathcal{N}, \mathcal{M}, !N$.

w Soit $f : \uparrow 1 \longrightarrow [\![\mathcal{N}]\!]$ interprétant $\vdash \mathcal{N}$. On construit $(\lambda^{\parr}_{\uparrow 1})^{-1}; w_P \parr f : \uparrow 1 \longrightarrow ?[\![P]\!] \parr [\![\mathcal{N}]\!]$ qui interprète $\vdash \mathcal{N}, ?P$.

$\bar{w}$ On construit $\text{refl}; \bar{w}_{N^\perp} : (![\![N]\!])^{\perp_L} \longrightarrow \perp$ interprétant $\vdash !N$.

d Soit $f : [\![P]\!]^{\perp_L} \longrightarrow [\![\mathcal{M}]\!]$ interprétant $\vdash P, \mathcal{M}$. On construit

$$\chi(\uparrow(1 \otimes \text{clos}^{-1}_{[\![P]\!]}); \uparrow \lambda^{\otimes}_{\downarrow [\![P]\!]^{\perp_L}}; \text{compl}; f); d_{[\![P]\!]} \parr [\![\mathcal{M}]\!]$$

interprète une preuve de $\vdash \mathcal{M}, ?P$.

$\bar{d}$ Soit $f : \uparrow 1 \longrightarrow [\![N]\!] \parr [\![\mathcal{M}]\!]$ interprètant une preuve de $\vdash \mathcal{M}, N$. On définit

$$f' := \uparrow \lambda^{-1}_{[\![N]\!]^{\perp_R}}; \chi^{-1}_{[\![N]\!]}(f)$$

et on construit $\text{refl}; \bar{d}_{[\![N]\!]^{\perp_R}}; f' : (![\![N]\!])^{\perp_L} \longrightarrow [\![\mathcal{M}]\!]$ qui interprète une preuve de $\vdash \mathcal{M}, !N$.

4.4 Invariance par élimination des coupures

Nous démontrons que l'interprétation d'une preuve après l'élimination d'une coupure est la même qu'avant l'élimination modulo l'utilisation de l'adjonction χ et de son inverse χ^{-1} et des isomorphismes suivants : $\lambda, \rho, \gamma, \alpha$, clos, refl ainsi que de leurs inverses. Beaucoup des résultats qui suivent proviennent d'outils fournit par le biproduit [18]. La plupart des démonstrations utilisent la naturalité des isomorphismes des catégories monoïdales symétrique et certains diagrammes. Les cas co-déréliction/contraction et co-contraction/déréliction sont les cas les plus exigeants : leur démonstration utilise en particulier la compatbilité du biproduit avec $⅋$ et des diagrammes du biproduit.

Nous commençons par quelques lemmes préliminaires. Le lemme suivant se démontre à l'aide du fait que J est terminal et initial (respectivement).

Lemme 28. Soit f un morphisme de A' vers A, alors on a :

$$f; 0_{C(A,B)} = 0_{C(A',B)}$$

Soit g un morphisme de B vers B', alors on a :

$$0_{C(A,B)}; g = 0_{C(A,B')}$$

Les démonstrations des propriétés qui suivent font intervenir les notions de naturalité, de produit cartésien et des diagrammes de catégories monoïdales. Elles serviront plus tard pour l'invariance de la sémantique par l'élimination des coupures pour les cas contraction/co-déréliction et déréliction/co-contraction.

Proposition 29. *On a l'égalité :*

$$\Delta^{\&}_{?P⅋?P}; (\bar{d}_{\uparrow P} ⅋ \bar{w}_{?P}) \,\&\, (\bar{w}_{?P} ⅋ \bar{d}_{\uparrow P}); \rho^{⅋} ⅋ \lambda^{⅋}; \nabla^{\&}_{\uparrow P} = (m^{P,P}_{?,⅋,\oplus})^{-1}; ?\nabla_P; \bar{d}$$

Proposition 30. *On a l'égalité :*

$$\Delta^{\&}; \rho^{-1} \,\&\, \lambda^{-1}; (d ⅋ w) \,\&\, (w ⅋ d); \nabla^{\&} = d; ?\Delta; m^{P,P}_{?}$$

Cette égalité se montre avec les mêmes arguments que la preuve de l'égalité précédente, en utilisant le fait que le biproduit $\&$ est un produit co-cartésien cette fois-ci.

Théorème 31. *L'axiomatique donnée par la définition 17 forme un modèle dénotationnel de la logique linéaire différentielle finitaire et polarisée.*

5 Conclusion

Nous avons construit une axiomatique catégorique pour la logique linéaire différentielle, finitaire polarisée. Celle-ci est construite sur la notion de chiralité, de biproduit et de co-déréliction. Notre but était de mettre à jour au niveau catégorique la facilité avec laquelle la polarisation apparaît dans les modèles de la logique linéaire différentielle.

Nous n'avons que partiellement réussi : en effet, l'ajout d'une co-déréliction se fait toujours de manière ad-hoc, et ne rend pas compte du caractère dual entre la déréliction et la co-déréliction. De plus, la nécessité de demander un biproduit sur chacune des deux catégories de la chiralité est particulièrement lourde, et il faudrait dans l'idéal pouvoir s'en dégager. Nous pensons qu'exiger un biproduit sur $\mathscr{N}$ devrait suffire - cela donnerait naturellement un coproduit sur $\mathscr{P}$. Il reste néanmoins à vérifier les preuves dans le détail.

On remarque par ailleurs que quand la partie linéaire de DiLL polarisée est interprétée par une chiralité, la partie exponentielle ressemble elle aussi à une chiralité. En effet, la partie exponentielle est interprétée elle aussi par une clôture monoïdale forte (entre $\mathscr{E}$ et $\mathcal{U}$), ainsi que par des morphismes $\bar{d}$ et d se comportant comme des shifts entre $\mathscr{E}$ et $\mathcal{U}$. Nous conjecturons que l'interprétation de DiLL *avec promotion* se fera via deux chiralités, l'une traitant de l'interaction positif / négatif et l'autre de l'interaction linéaire / non-linéaire. Nous espérons qu'une telle structure catégorique permettra de décomposer la différentiation avec des opérateurs plus primitifs, et d'éventuellement mettre en valeur les opérateurs de contrôle en jeu dans le calcul de la différentielle.

Références

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. 1992.

[2] R. F. Blute, J. R. B. Cockett, and R. A. G. Seely. Differential categories. *Math. Structures Comput. Sci.*, 16(6), 2006.

[3] T. Ehrhard and L. Regnier. Differential interaction nets. *Theoretical Computer Science*, 364(2) :166–195, 2006.

[4] Thomas Ehrhard. On köthe sequence spaces and linear logic. *Mathematical Structures in Computer Science*, 12(5) :579–623, 2002.

[5] Thomas Ehrhard. Finiteness spaces. *Mathematical Structures in Computer Science*, 15(4) :615–646, 2005.

[6] Thomas Ehrhard. An introduction to differential linear logic : proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7) :995–1060, 2018.

[7] Thomas Ehrhard and Olivier Laurent. Interpreting a finitary pi-calculus in differential interaction nets. *Inf. Comput.*, 2010.

[8] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1) :1–102, 1987.

[9] Jean-Yves Girard. A new constructive logic : Classical logic. *Mathematical Structures in Computer Science*, 1991.

[10] A. Grothendieck. Produits tensoriels topologiques et espaces nucléaires. *Mem. Amer. Math. Soc.*, No. 16 :140, 1955.

[11] Hans Jarchow. *Locally convex spaces*. B. G. Teubner, 1981.

[12] M. Kerjean and C. Tasson. Mackey-complete spaces and power series. *MSCS*, 2016.

[13] Marie Kerjean. A logical account for linear partial differential equations. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford*.

[14] Marie Kerjean and Jean-Simon Pacaud Lemay. Higher-order distributions for differential linear logic. In *FOSSACS 2019, Held as Part ETAPS, Prague, Czech Republic, Proceedings*.

[15] O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, March 2002.

[16] P.-A. Melliès. Categorical semantics of linear logic. *Société Mathématique de France*, 2008.

[17] Paul-André Melliès. Dialogue categories and chiralities. *Publ. Res. Inst. Math. Sci.*, 52(4) :359–412, 2016.

[18] M.Fiore. Differential structure in models of multiplicative biadditive intuitionistic linear logic. *Proceedings of TLCA*, 2007.

[19] L. Schwartz. *Théorie des distributions*. Publications de l'Institut de Mathématique de l'Université de Strasbourg, No. IX-X. Nouvelle édition, entiérement corrigée, refondue et augmentée. Hermann, Paris, 1966.

[20] Lionel Vaux. Differential linear logic and polarization. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 371–385, 2009.

Vers la formalisation en Coq des transformateurs de monades modulaires

Célestine Sauvage[1], Reynald Affeldt[2], and David Nowak[1]

[1] CRIStAL *, Lille, France
[2] National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Japan

Résumé

Nous étudions la vérification formelle de programmes avec effets de bord en utilisant un langage purement fonctionnel. Dans le cadre de cette étude, nous avons développé Monae, une librairie Coq qui propose une formalisation des monades et de leurs lois algébriques. Les preuves se font par raisonnement équationnel en utilisant les capacités de réécriture de Coq. Les programmes n'utilisent généralement pas un seul type d'effet de bord, mais une combinaison de plusieurs d'entre eux. On utilise les transformateurs de monades dans ce but. Cependant, l'approche traditionnelle pour le *lifting* des primitives n'est pas modulaire. Il est intéressant de définir de manière canonique les opérations algébriques des monades et leurs primitives `lift`. Dans cet article, nous présentons l'implémentation des transformateurs de monades modulaires et les preuves des théorèmes qui en découlent en Coq. Nous montrons également leurs utilisations comparées aux transformateurs de monades classiques.

1 Introduction

La programmation fonctionnelle utilise des monades pour représenter les différents types d'effets de bord existant dans les programmes impératifs, tels que les états, les exceptions ou les continuations. Chaque monade s'accompagne d'opérations qui lui sont propres et ces opérations doivent respecter certaines conditions de cohérence. Les programmes utilisant généralement plusieurs sortes d'effets de bord, on voudrait pouvoir en combiner les monades correspondantes. Cependant, obtenir une monade qui modélise cette combinaison de plusieurs effets peut être difficile. Certaines méthodes ont vu le jour pour pallier ce problème.

Dans un premier temps, une manière de composer des monades a été étudiée dans Monae suivant l'idée de Moggi [6]. Mais cela implique de devoir réécrire un modèle de monade pour chaque combinaison [1]. Une autre approche plus modulaire a ensuite vu le jour, les *transformateurs de monades* [5]. Un transformateur de monades ajoute, pour toute monade, un nouvel effet tout en assurant que l'effet initial soit toujours effectif à l'aide de la primitive `lift`.

Malgré tout, le *lifting* des opérations de la monade sous-jacente a besoin d'être spécifié de manière ad hoc, pour chaque transformateur de monades, ce qui n'est pas modulaire. De plus, le nombre de *lifting* des primitives augmente avec le nombre de monades combinées. Pour améliorer cette modularité, nous formalisons en Coq les transformateurs de monades modulaires [4].

Dans cet article, nous décrivons l'état actuel de cette formalisation [1]. Nous présenterons dans un premier temps Monae et son fontionnement (section 2), puis nous montrerons comment nous avons ajouté les transformateurs de monades dans Monae (section 3). Nous terminerons enfin en nous focalisant sur le *lifting* modulaire des opérations liées aux monades (section 4).

*Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

1. Le code et les preuves décrits dans ce papier sont publics et disponibles à l'adresse suivante : https://github.com/affeldt-aist/monae.

2 Préliminaires techniques

Monae est une librairie qui formalise une hiérarchie de monades et leurs effets [1]. Dans cette hiérarchie, les monades sont des endo-foncteurs dans une catégorie particulière : les objets sont les types qui ont le type `Type` de Coq et les morphismes sont les fonctions de type `Type -> Type`. La base de cette hiérarchie est le type `functor`. Étant donné un foncteur `F`, on note `F A` l'application de `F` au type `A` et `F # g` l'application de `F` à la fonction `g`. Les monades sont représentées par un type `monad` qui étend le type `functor` en l'équipant avec deux transformations naturelles (`Ret` et `Join`) et les lois de monade usuelles. On définit aussi l'opération `Bind m f` (notée $m \gg\!= f$ dans les exemples de code) à partir de `Ret` et `Join`. On peut ainsi définir les monades à partir de `Ret` et `Bind` également. La hiérarchie de Monae est construite en étendant le type `monad` avec des interfaces qui définissent les effets sous la forme d'opérations et de lois équationnelles comme préconisé par Gibbons et Hinze [3]. Les interfaces peuvent être combinées, et les modèles sont construits de manière ad hoc dans un deuxième temps.

Exemple : La monade d'état

Pour illustrer cette construction, nous présenterons ici la monade d'état définie dans Monae[2]. Les monades sont formalisées en utilisant la technique des *packed classes* [2]. Dans un premier temps, on définit une interface pour la monade d'état (`mixin_of`) qui étend la classe `monad` (`class_of`). Le résultat est empaqueté avec une fonction de type `Type -> Type` qui représente l'action sur les objets de la monade :

```
Record mixin_of S (M : monad) : Type := Mixin {
  (* Operations *)
  get : M S ;
  put : S -> M unit ;
  (* Équations *)
  _ : forall s s', put s >> put s' = put s' ;
  _ : forall s, put s >> get = put s >> Ret s ;
  _ : get >>= put = skip ;
  _ : forall A (k : S -> S -> M A),
      get >>= (fun s => get >>= k s) = get >>= fun s => k s s }.
Record class_of S (m : Type -> Type) := Class {
  base : Monad.class_of m ; mixin : mixin_of S (Monad.Pack base) }.
Structure t S : Type := Pack { m : Type -> Type ; class : class_of S m }.
```

La cohérence des propriétés est vérifiée dans un deuxième temps en implémentant le modèle classique pour la monade d'état (voir section 3.2 pour plus de détails sur la validation d'un modèle dans Monae).

3 Étendre Monae avec les transformateurs de monades

Comme expliqué précédemment, un transformateur de monades peut être utilisé pour composer des effets de manière modulaire, produisant une couche d'effets monadiques élémentaires. Un transformateur de monades est un constructeur de type `T` prenant en argument une monade `M`, de telle sorte que `T M` soit elle aussi une monade. Pour pouvoir faire le *lifting* de l'effet

2. fichier `state_monad.v`

monadique de M, le transformateur de monades est équipé d'une primitive, lift, qui satisfait les lois suivantes :

$$\begin{aligned} \texttt{lift} \circ \texttt{Ret}_{\texttt{M}} &= \texttt{Ret}_{\texttt{T M}} \\ \texttt{lift}(m \gg\!=_{\texttt{M}} k) &= \texttt{lift}(m) \gg\!=_{\texttt{T M}} (\texttt{lift} \circ k) \end{aligned}$$

Comme les opérations peuvent se combiner de différentes manières suivant le transformateur de monades au-dessus, il est nécessaire d'exprimer manuellement le *lifting* de ces opérations à travers le transformateur. On définit les transformateurs de monades dans la section 3.1 qu'on illustre avec l'exemple du transformateur de monades d'erreur dans la section 3.2.

3.1 Morphisme de monades et transformateur de monades

On formalise [3] les transformateurs de monades en suivant la présentation de Jaskelioff [4] qui définit d'abord les morphismes de monades.

Soient deux monades M and N. Un *morphisme de monades* est une fonction e de type forall A, M A -> N A (on écrit M ~~> N par la suite) telle que, pour tout type A, Ret = $\texttt{e}_{\texttt{A}} \circ$ Ret et, pour tout types A et B, et m et f de types appropriés, $\texttt{e}_{\texttt{B}}(\texttt{m} \gg\!= \texttt{f}) = \texttt{e}_{\texttt{A}}\texttt{m} \gg\!= (\texttt{e}_{\texttt{B}} \circ \texttt{f})$:

```
(* dans le Module monadM *)
Record class_of (e : M ~~> N) := Class {
  _ : forall A, Ret = e A \o Ret;
  _ : forall A B (m : M A) (f : A -> M B),
    e B (m >>= f) = e A m >>= (e B \o f) }.
Structure t := Pack { e : M ~~> N ; class : class_of e }.
```

Un *transformateur de monades* est une fonction T de type monad -> monad (pour toute monade M, T M est donc une monade) munie d'une opération lifT telle que liftT M est un morphisme de monades :

```
(* dans le Module MonadT *)
Record class_of (T : monad -> monad) := Class {
  liftT : forall M : monad, monadM M (T M) }.
Record t := Pack {m : monad -> monad ; class : class_of m}.
```

3.2 Exemple : le transformateur de monades d'erreur

Le transformateur de monades d'erreur [4] ajoute à une monade M existante un effet qui permet à un programme d'échouer et de retourner une erreur de type Z. Pour définir ce transformateur de monades, on va devoir fournir :

1. une fonction de type monad -> monad ;
2. un morphisme de monades pour définir liftX.

Considérons le type de l'erreur Z et une monade M comme donnés. L'action sur les objets de la nouvelle monade est :

```
Definition MX := fun X => M (Z + X).
```

3. fichier monad_transformer.v
4. fichier monad_transformer.v

On montre que l'action sur les morphismes `MX_fmap` satisfait les lois des foncteurs ce qui nous permet de construire un foncteur `MX_functor` (`fmap f` est une notation pour `_ # f` où le foncteur est inféré automatiquement) :

```
Definition MX_map A B (f : A -> B) (m : MX A) : MX B :=
  fmap (fun x => match x with inl y => inl y | inr y => inr (f y) end) m.
Definition MX_functor : functor. (* lois des foncteurs omises *)
```

Les définitions de `retX` et `bindX` sont celles usuelles de la définition du transformateur de monades d'erreur, et où le `Ret` et le `Bind` (`>>=`) utilisés sont ceux de la monade `M` :

```
Definition retX X x : MX X := Ret (inr x).
Definition bindX X Y (t : MX X) (f : X -> MX Y) : MX Y :=
  t >>= fun c => match c with inl z => Ret (inl z) | inr x => f x end.
```

On prouve que `retX` est une transformation naturelle du foncteur identité `FId` vers le foncteur `MX_functor` (construction `retX_natural` omise), ce qui nous permet de construire une nouvelle monade en utilisant le constructeur `Monad_of_ret_bind` de Monae :

```
Definition eErrorMonadM : monad :=
  @Monad_of_ret_bind MX_functor retX_natural bindX _ _ _ (* lois omises *).
```

Il ne nous reste plus qu'à fournir la fonction `liftX` qui transforme toute computation `M X` en une computation `eErrorMonadM X` :

```
Definition liftX X (m : M X) : eErrorMonadM X :=
        @Bind M _ _ m (fun x => @Ret eexceptionMonadM _ x).
```

Comme cette dernière satisfait les lois de morphismes de monades (construction `ErrorMonadM` omise), on obtient finalement un transformateur de monades

```
Definition errorMonadM : monadM M eErrorMonadM :=
  monadM.Pack (@monadM.Class _ _ liftX _ _).
```

4 Opérations et *lifting*

On montre dans cette section que la formalisation de transformateur de monades de la section précédente permet de retrouver la preuve du premier théorème de [4] au moyen d'une preuve succincte à base de réécriture.

4.1 Opérations et opérations algébriques

Comme évoqué au-dessus, en plus des opérations de base `Ret` et `Bind`, on associe aux monades des opérations pour représenter des effets. Pour parler de *lifting* des opérations par morphisme de monades, Jaskelioff [4] distingue les *(E,M)-opérations* et les *opérations algébriques* (les opérations usuelles peuvent être retrouvées à partir des (`E`,`M`)-opérations).

Étant donnés un foncteur `E` et une monade `M`, on appelle **(`E`,`M`)-opération** une transformation naturelle de `E \O M` vers `M` (où `\O` est la composition de foncteurs).

Une (`E`,`M`)-opération `op` est algébrique quand, pour tout `A`, `B`, `t` et `f`, on a

```
(op A t >>= f) = op B ((E # (fun m => m >>= f)) t).
```

Soient un morphisme de monades `e` de `M` vers `N`, un foncteur `E` et une (`E`,`M`)-operation `op`. Un *lifting* de `op` vers `N` le long de `e` est une (`E`,`N`)-operation `op'` telle que :

```
forall X, e X \o op X = op' X \o (E # (e X)).
```

4.2 Exemple : Opération algébrique put

Soit S un type pour un état. D'après [4, section 3], on définit [5] le foncteur put_fun tel que :

```
(* Foncteur pour l'opération put (action sur les objets) *)
Definition put_acto X := (S * X).
(* Map associée au foncteur (action sur les morphismes) *)
Definition put_actm X Y (f : X -> Y) (sx : put_acto X) : put_acto Y :=
 (sx.1, f sx.2).
(* Pack dans la classe Functor *)
Program Definition put_fun :=
  Functor.Pack (@Functor.Class put_acto put_actm _ _).
(* Preuves des lois de foncteurs omises *)
```

On vérifie ensuite que l'on a bien une transformation naturelle de put_fun S \O MS S vers MS S (où MS est le transformateur de monades d'état) :

```
Definition n_put S A (s : S) (m : MS S A) : MS S A :=
  fun _ => m s.
Lemma naturality_put S : naturality (put_fun S \O MS S) (MS S)
  (fun A => uncurry (n_put (A:=A))).
```

On montre que l'on obtient bien une opération algébrique suivant la définition donnée dans la section 4.1 :

```
Definition put_op S : operation (put_fun S) (ModelMonad.State.t S) :=
  Natural.Pack (@naturality_put S).
Lemma algebraic_put S : algebraicity (put_op S). Proof. ... Qed.
Program Definition put_aop S : aoperation (put_fun S) (MS S) :=
  AOperation.Pack (AOperation.Mixin (@algebraic_put S)).
```

L'opérateur put peut être défini à partir de l'(E,M)-opération put_op et a la même sémantique que l'opération usuelle de la littérature :

```
Definition put : S -> MS S unit := fun s => put_op _ (s, Ret tt).
Lemma putE : put = fun s' _ => (tt, s'). Proof. by []. Qed.
```

4.3 *Lifting* des opérations algébriques

Le premier théorème que Jaskelioff prouve à propos du *lifting* des opérations montre que les opérations algébriques sont transformées en opérations algébriques par morphisme de monades.

Theorem 1 (*Lifting* algébrique)**.** *Soit la (E,M)-opération* `op`*. La (E,N)-opération définie par*

```
fun X => Join \o e (N X) \o phi op (N X)
```

où `phi op` *est*

```
fun X => op X \o (E # Ret)
```

est algébrique [4, Théorème 19].

5. fichier monad_model.v

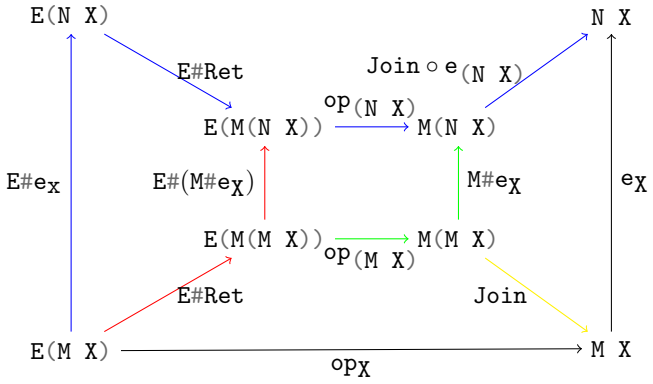


FIGURE 1 – Preuve du théorème [4, Théorème 19]

Démonstration. [6] Par définition, le but original est :

```
forall Y, e X (op X Y) = Join (e (N X) (op (N X) ((E # Ret) ((E # e X) Y)))).
```

Le membre gauche correspond au chemin noir de la figure 1 et le membre de droite correspond au chemin bleu. La première étape est de montrer que la première partie du chemin bleu commute avec le chemin rouge, c'est-à-dire :

```
(E # Ret) ((E # e X) Y) = (E # (M # e X)) ((E # Ret) Y).
```

La preuve consiste en une série de réécritures. On utilise d'abord la loi de compositions du foncteur `E` dans les deux membres de l'équation, puis la naturalité de `Ret` de la monade `M`, et enfin une propriété du foncteur identité :

```
rewrite -[in LHS]compE -functor_o.
(* (E # (Ret \o e X)) Y = (E # (M # e X)) ((E # Ret) Y) *)
rewrite -[in RHS]compE -functor_o.
(* (E # (Ret \o e X)) Y = (E # (M # e X \o Ret)) Y *)
rewrite (natural RET).
(* (E # (Ret \o e X)) Y = (E # (Ret \o FId # e X)) Y *)
by rewrite FIdf.
```

Ensuite, on montre qu'une partie du chemin rouge-bleu commute avec le chemin vert :

```
op (N X) (E # (M # e X)) = (M # e X) (op (M X)).
```

Puis on montre que le chemin vert-bleu commute avec le chemin vert-jaune-noir :

```
e X (Join (op (M X))) = Join (e (N X) ((M # e X) (op (M X)))).
```

Toutes les étapes de preuve jusqu'à présent sont affaire de réécriture des lois de foncteurs et de monades. La dernière étape est l'égalité entre le chemin rouge-vert-jaune et `op X Y` et repose sur un lemme intermédiaire [4, Proposition 17]. □

6. fichier `monad_transformer.v`

4.4 Exemple : Lifting des opérateurs `get` et `put`

On voudrait écrire un programme qui utilise à la fois la monade option (un cas particulier de la monade d'erreur avec `unit` le type de retour de l'erreur) et la monade d'état. On comparera le *lifting* classique des (`E`,`M`)-opérations algébriques de la monade d'état et le *lifting* algébrique de ces mêmes opérations.

4.4.1 *Lifting* classique

Il faut préciser la manière de faire le *lifting* des deux opérations dans le transformateur de monades d'option :

```
Let M := stateMonad.
Let erZ : monadT := errorT unit.

Definition getOpt : erZ (M S) := liftX get.
Definition putOpt : erZ (M S) := fun s' => liftX (put s').
```

On peut maintenant écrire un programme mélant les effets de la monade option et de la monade d'état :

```
Let incr := getOpt >>= (fun i => putOpt (i.+1)).
Let prog := incr >> Fail >> incr.
```

`Fail` étant une opération de la monade d'erreur.

4.4.2 *Lifting* algébrique

Suivant la définition donnée pour le *lifting* algébrique (`alifting`) de la section 4.3, Théorème 1, on obtient les (`E`,`N`)-opérations suivantes (où `N` correspond à une monade résultant de l'application du transformateur de monades d'erreur) :

```
Let lift_getX S : (get_fun S) \O (erZ (M S)) ~~> (erZ (M S)) :=
  alifting (get_aop S) (LiftT erZ (M S)).
Let lift_putX S : (put_fun S) \O (erZ (M S)) ~~> (erZ (M S)) :=
  alifting (put_aop S) (LiftT erZ (M S)).
(* LiftT erZ (M S) correspondant ici à liftX *)
```

On peut à présent écrire des programmes avec les (`E`,`N`)-opérations (en respectant l'utilisation de ces opérations décrites dans [4, section 3] pour que leurs sémantiques soient identiques au `get` et `put` usuels).

```
Let incr := (lift_getX Ret) >>= (fun i => lift_putX (i.+1, Ret tt)).
Let prog := incr >> Fail >> incr.
```

On peut comparer que le *lifting* algébrique et le *lifting* classique ont le même comportement :

```
Goal lift_getX Ret = @liftX  _ _ _ get.
Proof. by rewrite /lift_getX aliftingE. Qed.

Goal (fun s' => (lift_putX (s', Ret tt))) =
     fun s => (@liftX _ _ _ (put s)).
Proof. by rewrite/lift_putX aliftingE. Qed.
```

5 Conclusion et travaux futurs

Dans cet article, nous avons donné un aperçu de Monae, une formalisation en Coq d'une hiérarchie de monades pour le raisonnement équationnel [1] et expliqué comment l'étendre avec des transformateurs de monades. Pour ce faire, nous avons formalisé une proposition existante qui insiste sur les aspects modulaires [4]. Monae semble se prêter naturellement à cette extension : nous avons pu reproduire les définitions (à propos des effets pour les états, les exceptions, mais aussi les continuations) et les premières preuves de Jaskelioff en utilisant la réécriture (voir en particulier section 4.3). Nous avons pu développer quelques exemples sur différents transformateurs de monades et leurs opérations algébriques.

Nous comptons maintenant terminer l'expérience de formalisation de la proposition de Jaskelioff, qui définit aussi le *lifting* d'opérations non-nécessairement algébriques [4, section 5], et utiliser les transformateurs de monades pour le raisonnement équationnel à la Gibbons et Hinze [3]. Dans ce dernier cas, il nous faudra généraliser les interfaces des monades pour y accommoder les transformateurs de monades.

Références

[1] Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7-9, 2019*, volume 11825 of *Lecture Notes in Computer Science*. Springer, 2019.

[2] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[3] Jeremy Gibbons and Ralf Hinze. Just do it : simple monadic equational reasoning. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM, 2011.

[4] Mauro Jaskelioff. Modular monad transformers. In *18th European Symposium on Programming (ESOP 2009), York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2009.

[5] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*, pages 333–343. ACM, 1995.

[6] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1) :55–92, 1991.

Étude formelle de l'implémentation du code des impôts*

Denis Merigoux[1], Raphaël Monat[2] et Christophe Gaie[3]

[1] Inria, Centre de Paris, 12 rue Simone Iff, 75012, Paris, France
denis.merigoux@inria.fr
[2] LIP6, Sorbonne Université – CNRS, 4 place Jussieu, 75005 Paris, France
raphael.monat@lip6.fr
[3] Direction Générale des Finances Publiques, Bureau SI-1E, Le Montaigne, Noisy-Le-Grand
christophe.gaie@dgfip.finances.gouv.fr

Résumé

Le code des impôts définit dans son texte législatif une fonction mathématique permettant de calculer l'impôt sur le revenu d'un foyer fiscal. Afin de recouvrer l'impôt, cette fonction est implémentée sous la forme d'un algorithme par la Direction Générale des Finances Publiques (DGFiP), en utilisant un langage dédié appelé M (pour macro-langage). Nous proposons une sémantique formelle du langage M, testée grâce aux données publiées par la DGFiP. Cette formalisation, couplée à la publication par la DGFiP de la base de code M calculant l'impôt, nous donne accès à une formalisation complète de la portion du code des impôts définissant l'algorithme de calcul de l'impôt sur le revenu. Nous démontrons l'utilité d'une telle formalisation grâce à un prototype à base de solveurs SMT permettant d'inférer des méta-propriétés sur le calcul de l'impôt. Ces méta-propriétés peuvent ensuite compléter et affiner les analyses économiques existantes sur les effets redistributifs de l'impôt sur le revenu, mais aussi de diverses allocations. Plus généralement, une formalisation systématique des portions algorithmiques de la loi permettrait d'augmenter le niveau d'assurance sur la cohérence du système socio-fiscal français.

Trois artefacts logiciels accompagnent cet article : une formalisation mécanisée de la sémantique du langage M, un compilateur pour le langage M basé sur cette sémantique, ainsi que le prototype d'encodage du code des impôts dans le solveur SMT Z3.

1 Quelle implémentation pour le système fiscal français ?

Le principe de tout impôt est de prélever pour le compte de l'État une partie d'un flux monétaire ou d'un capital. Mais de quelle taille est cette partie ? Comment calculer le montant de l'impôt à partir des paramètres de l'individu, du ménage ou de l'entreprise qui y est soumise ? La réponse se trouve dans la loi, et en France tout particulièrement dans le code général des impôts. De ce fait, les lois définissant le montant de l'impôt évoquent des quantités chiffrables et mesurables : revenus, nombre de personnes à charge, etc. L'impôt est donc défini par une fonction mathématique f, qui associe à chaque ménage ou entreprise x (défini par les caractéristiques prévues dans la loi) un impôt unique $f(x)$.

Comment l'impôt est-il calculé en pratique ? Tout dépend de la complexité de la fonction f. Pour le cas d'impôts forfaitaires ou proportionnels à une somme d'argent, f est très simple et la valeur $f(x)$ peut être calculée de tête. Mais le principe de progressivité de l'impôt sur le revenu impose une fonction f plus complexe ; il faut alors poser le calcul à partir de la déclaration de revenus. Avec l'arrivée de l'informatique, le calcul est automatisé et programmé

*Ce travail a été en partie financé par le Conseil Européen de la Recherche dans le cadre de la bourse "Consolidator Grant" 681393 - MOPSA.

sur ordinateur ; il existe donc nécessairement un programme informatique, appelé P, dont la tâche est de calculer $f(x)$ à partir de x. P et f sont équivalents dans un certain sens, puisqu'ils calculent la même quantité.

On reconnaît là une problématique récurrente des méthodes formelles : la correspondance entre une spécification (f) et une implémentation (P). Le but de cet article est donc d'appliquer certains résultats classiques du champ des méthodes formelles au couple f et P qui définit le calcul des impôts, mais aussi au reste du système socio-fiscal français : cotisations et allocations. Même si la méthode peut être appliquée à n'importe quel pays recouvrant des impôts, la France fait partie des pays où le calcul du montant de l'impôt est à la charge de la puissance publique à partir des déclarations fiscales. Il est donc crucial que le programme P soit la transcription exacte de la fonction f définie par la loi, sous peine de recours légaux des contribuables lésés par une erreur de calcul.

Les contributions de cet article sont les suivantes :

— une présentation et formalisation avec preuve de sûreté du typage de M, le langage utilisé par la Direction Générale des Finances Publiques (DGFiP) pour définir le programme P de calcul de l'impôt sur le revenu ;

— cette présentation est accompagnée d'une définition en Coq de la sémantique et d'une preuve de sûreté du typage, ainsi que d'un compilateur permettant d'exécuter le code du calcul des impôts ;

— un prototype basé sur un solveur SMT pour l'analyse de propriétés du système socio-fiscal.

La section 2 présente la vision ainsi que le cadre d'utilisation de M au sein de la DGFiP. La section 3 analyse le langage M sous un angle formel. Dans la section 4, on explore ensuite diverses pistes d'application découlant de la formalisation. Enfin, la section 5 remet ce travail dans le contexte des initiatives ayant pour objectif la formalisation de la loi.

2 Le langage M : historique et utilisation à la DGFiP

2.1 Présentation du calcul de l'impôt sur le revenu

Le processus de gestion de l'impôt sur le revenu (noté dans la suite IR) nécessite de calculer le montant d'impôt d'une situation fiscale dans trois grands types de situations :

— le calcul primitif permet de connaître le montant d'impôt dû sur les revenus de l'année précédente. Cette modalité de calcul peut être utilisée pour réaliser une simulation d'impôt, pour informer le contribuable ou pour fixer le montant d'impôt dû par le contribuable (en termes légaux, pour "figer le montant d'impôt dû par le contribuable dans le cadre du rôle [1] d'imposition à l'impôt sur le revenu").

— le calcul correctif permet de modifier le calcul primitif. Cette modalité peut être notamment utilisée lors d'un contentieux au bénéfice ou au détriment d'un contribuable. Il s'applique aux revenus déclarés pour l'année $N-1$.

— le calcul de modulation permet de recalculer le taux de prélèvement à la source en cours d'année. Ce calcul fait généralement suite à une modification de situation de famille (mariage, divorce, naissance, décès, ...) ou à une évolution des revenus du foyer fiscal (perte d'emploi, reprise d'activité, nouveaux revenus fonciers, ...). Cette modalité s'applique aux revenus contemporains de l'année N (vision prospective des revenus qui seront déclarés). Un dispositif anti-abus est également prévu.

1. Le rôle d'imposition est le titre exécutoire en vertu desquels les comptables publics effectuent et poursuivent le recouvrement de l'impôt sur le revenu.

Au vu des différentes modalités de calcul, mais également de la diversité des applications qui doivent les réaliser, la « calculette IR » a été créée en 1990. Elle permet d'assurer l'unicité et la cohérence du calcul par de multiples applications. Elle offre également un gain d'efficience significatif puisqu'une seule équipe est en charge du développement de ce module qui gère plus de 4000 variables et plus de 1000 règles de taxation. Le code M publié comporte ainsi environ 92 000 lignes de code pour le millésime des revenus 2017. La réalisation d'un module de calcul commun en lieu et place d'un développement spécifique permet de réaliser un gain d'efficience d'un facteur 5 environ (1 équipe réalise le travail de 6 ou 7 autres qui doivent néanmoins embarquer le composant mutualisé).

Voici quelques informations concernant la publication du code source :

— la première publication a eu lieu dans le cadre du hackhation #CodeImpot qui s'est déroulé en avril 2016 [2] ;

— le dépôt officiel du code source est `https://framagit.org/dgfip/ir-calcul` ;

— ce dépôt dispose du code de taxation primitif des revenus 2010 à 2017 ;

— le prochain millésime de taxation (revenus 2018) contiendra les éléments ayant permis de calculer le taux de prélèvement à la source et de réaliser la modulation (en cas de changement de situation de famille et/ou de revenus).

2.2 Organisation des travaux de l'équipe en charge du calcul IR

Dès le mois de septembre, l'équipe développe simultanément le moteur de calcul primitif sur les revenus $N-1$ et celui pour la modulation des revenus N. Le choix d'implémentation retenu consiste à réutiliser le calcul de taxation pour la modulation. Seules les règles de gestion différentes sont réécrites. Ce fonctionnement permet de ne pas dupliquer le code dans un nouveau module de calcul et réduit donc l'effort de maintenance ainsi que les risques d'erreur.

Pour valider les développements, l'équipe s'appuie sur les jeux d'essai fournis par la maîtrise d'ouvrage [3]. Ceux-ci sont totalement fictifs [4] et permettent de valider les différentes règles de calcul. Les jeux s'enrichissent d'année en année, permettant ainsi d'augmenter la couverture des tests. Ils permettent en outre de réaliser des tests de non-régression dans une démarche de type intégration continue.

Le calendrier d'utilisation des différentes calculettes est illustré dans la figure 1. Quelques précisions :

— ILIAD est l'application de gestion de l'impôt sur le revenu dans les services des impôts des particuliers. Cette application est écrite en Oracle Forms.

— TéléIR est l'application qui permet aux usagers de déclarer leurs revenus en ligne. Cette application est écrite en Java.

— GestPart est l'application de gestion des situations particulières. Elle permet notamment de corriger les déclarations des usagers qui seraient en anomalie via des actions d'agents DGFiP. Cette application est écrite en Java.

— EDI-IR est l'application qui permet aux comptables de procéder à la déclaration pour le compte de leurs clients, directement depuis leurs progiciels comptables par échange de données informatisées. Cette application est écrite en Java.

2. `https://www.etalab.gouv.fr/retour-sur-le-hackathon-codeimpot`

3. Pour le calcul primitif et le calcul de modulation, la maîtrise d'ouvrage est le bureau GF-1A en charge de l'animation de la fiscalité des particuliers. Pour le calcul correctif, c'est le bureau GF-1B en charge des applications d'assiette et de recouvrement forcé des impôts des particuliers.

4. Les jeux d'essais correspondent à des typologies de situations réelles mais aucune donnée n'est réelle puisque l'administration respecte absolument le secret fiscal garanti par la loi.

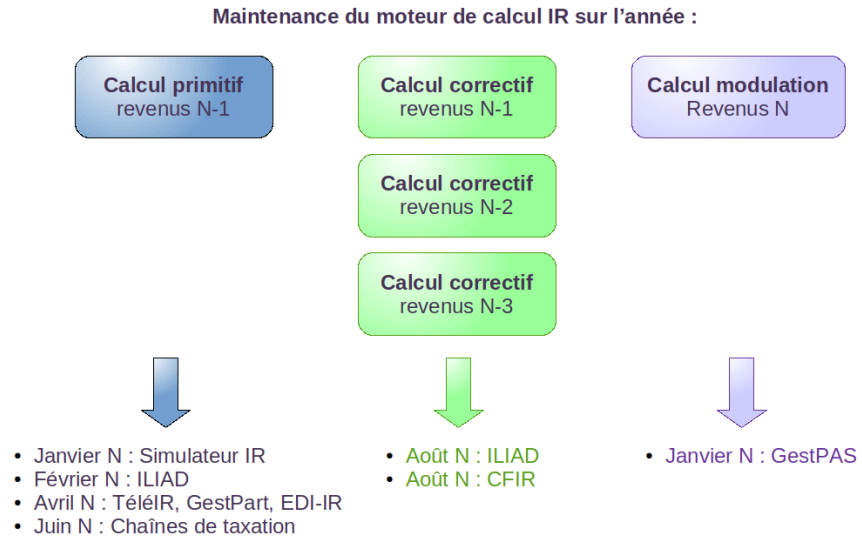


FIGURE 1 – Utilisations de la calculette IR

— Les chaînes de taxation sont les applications COBOL qui permettent notamment de calculer l'impôt sur le revenu, éditer l'avis d'imposition et établir le rôle d'imposition en vue de permettre le recouvrement.

— CFIR est l'application qui gère les conséquences financières à l'impôt sur le revenu. Elle permet le calcul et l'édition des droits et pénalités consécutives à des opérations de contrôle fiscal. Cette application est écrite en Visual Basic et Access.

— GestPAS est l'application qui permet d'effectuer les différentes opérations liées au prélèvement à la source. Cette application est écrite en Java.

Chaque application client embarque donc le moteur de calcul IR écrit dans le langage générique M. Ainsi la cohérence du calcul de l'impôt sur le revenu pour une version spécifique est garantie pour toutes les applications de la DGFiP. C'est en ce sens que nous pouvons dire que le langage M constitue un langage spécifique au domaine des finances publiques.

3 Étude formelle du langage M

Le calculateur du code des impôts écrit en langage M prend en entrée des paramètres numériques caractérisant un foyer fiscal au regard du fisc. Par exemple, la variable **1AJ** représente les revenus du premier déclarant, la variable **0AM** correspond au marriage d'un couple. Ces variables d'entrée sont soumises à des conditions de vérification dans le programme afin de modéliser des situations réalistes ; par exemple, **0AM** et **0AC** ne peuvent pas valoir 1 en même temps (on ne peut pas être marié et célibataire en même temps). Le calcul consiste ensuite en une suite d'affectation de variables (étant des scalaires ou des tableaux) à des expressions, et les opérations effectuées sont principalement des opérations arithmétiques, des conditionnelles et des calculs de seuil. Le langage ne possède pas de construction permettant d'exécuter des boucles de manière générale, il n'est donc pas Turing-complet. La sémantique de ce langage n'ayant pas été publiée par la DGFiP, nous avons dû faire un travail de rétro-ingénierie afin de l'établir. Par la suite, nous avons pu vérifier que notre sémantique était correcte en utilisant

la DGFiP comme oracle, ainsi qu'en utilisant certains fichiers de test. Le langage est assez dynamique, et cela est principalement dû à :

1. une valeur `indéfini` qui est utilisée pour dénoter les variables non initialisées ainsi que certaines situations souvent considérées comme des erreurs à l'exécution (telles que les divisions par zéro et les accès en dehors des bornes des tableaux) ;
2. cette valeur `indéfini` peut être convertie en des flottants via certaines règles de conversion (présentées en figure 6) ;
3. le langage possède des booléens mais en pratique, dans la base de code M publique, les opérations booléennes sont souvent réalisées à l'aide de l'arithmétique (0 pour faux, 1 pour vrai, l'addition comme « ou » logique, la multiplication comme « et », etc).

Nous commençons par présenter l'architecture générale du code de calcul de l'impôt, ainsi que la réduction vers un langage noyau légèrement simplifié, que nous appelerons par la suite « μM ». Nous présentons ensuite la syntaxe, le typage et la sémantique de ce langage noyau ; ces parties ont été formalisées en Coq. La dernière partie présente une implémentation *open-source* d'un compilateur pour le langage M d'origine, permettant de lire et exécuter le code de calcul mis à la disposition du public.

3.1 Réduction vers un langage noyau

Dans sa version du 6 octobre 2017, le code source du calculateur des impôts est séparé en 48 fichiers différents, qui totalisent 92 000 lignes. L'utilisation des variables est précédée d'une déclaration en tête de fichier, chaque variable étant associée à une description et un genre : une variable peut être « saisie » (c'est une entrée), « calculée », « restituée » (c'est une sortie). Des erreurs d'assertions, correspondant à des exceptions, sont aussi déclarées avec une description. Le code est ensuite séparé en des règles numérotées de manière unique (dont la numérotation n'influe pas l'ordre d'exécution). Chaque règle est annotée par une ou plusieurs applications, qui déclenchent son utilisation. Dans la suite, nous nous concentrons sur l'application `iliad`, présentée dans la section précédente. Les règles sont ensuite principalement des suites d'assignations, dont la syntaxe est décrite plus en détail en section 3.2. Les règles peuvent aussi contenir des assertions permettant de vérifier l'absence d'aberrations. La présentation effectuée ci-dessous se concentre sur le cœur du langage, omettant volontairement les facilités syntaxiques comme la notion de règle mentionnée précédemment, les boucles à itérations constantes utilisées pour définir les valeurs des tableaux, etc. De même, le cœur μM suppose que l'ordre d'exécution des assignations est fixé, tel que décrit dans le prochain paragraphe.

Ordre d'exécution Le code source ne présente pas d'ordre d'exécution apparent. L'outillage de la DGFiP semble faire une évaluation dynamique et paresseuse des variables en partant des variables de sorties qu'il faut calculer. Dans notre interpréteur OCaml, nous préférons exécuter une analyse de dépendance afin d'ordonner – partiellement – les assignations et assertions.

Avant d'exécuter un programme M, nous calculons donc le graphe de dépendance des variables, où $x \to y$ si et seulement si la variable x est utilisée dans la définition de la variable y. Il est également nécessaire d'inclure dans ce graphe les nœuds correspondant aux assertions. Ce graphe sera parcouru lors de l'exécution dans l'ordre topologique qui permet de garantir qu'en l'absence de cycles dans le graphe, si $x \to y$, alors l'assignation de x sera évaluée avant celle de y. En présence de cycles, il est nécessaire de découper le graphe en chacune de ses composantes fortement connexes avant de les évaluer dans l'ordre topologique ; nous utilisons l'algorithme de Tarjan [15] pour identifier ces composantes en temps linéaire. L'évaluation d'une

composante fortement connexe comportant plusieurs variables définies circulairement est faite en évaluant chacune des variables séparément en supposant que la valeur des autres variables de la composante fortement connexe est égale à `indéfini` dans un premier temps. Il est alors possible d'effectuer plusieurs passes d'évaluation, les suivantes prenant comme valeurs d'entrée pour les variables de la composante fortement connexe les valeurs calculées au tour précédent. Le nombre de passages est choisi arbitrairement lors de l'exécution, et il semble être de l'ordre de 10 en pratique. Ce calcul itératif est fortement utilisé par la DGFiP pour encoder des calculs de majorations d'impôts ou de correctifs qui ont une structure itérative. Dans la suite, nous ignorons cette notion de calcul itératif sur les composantes fortements connexes, puisque le code de ces calculs peut être expansé. Le programme ne possède pas d'entrées ni de sorties en tant que telles dans ce cœur du langage : les variables d'entrée sont simplement définies comme égales à leur valeur d'entrée (ou à `indéfini` s'il n'y a pas de valeur d'entrée), et il suffit de lire la valeur des variables de sortie après exécution du programme.

3.2 Syntaxe du langage μM

La base du langage μM est la définition de variables qui se fait grâce à un langage d'expressions très simple dont la syntaxe est présentée en figure 2. La figure 3 donne un exemple de programme μM se conformant à cette syntaxe.

Un programme $\langle$*programme*$\rangle$ est une liste de commandes $\langle$*commande*$\rangle$. Les commandes définissent la valeur des variables ainsi que des assertions conduisant à des erreurs lors de l'exécution du programme. Les variables peuvent être simples, ou bien correspondre à des tableaux de taille constante et connue à l'avance. Les valeurs simples sont des booléens ou des flottants. La valeur de chaque case du tableau est définie à l'aide d'une fonction d'un index générique `X`, variant entre les bornes de définition du tableau. Les expressions sont des variables (incluant l'index spécial de tableau appelé « `X` »), des litéraux, des combinaisons arithmétiques, logiques ou comparatives d'expressions, des conditionnelles, des appels de fonctions ou des accès dans des tableaux. Les fonctions sont fixées par le langage et ne peuvent pas être définies par un programme.

La fonction `arr` effectue un arrondi d'un flottant vers l'entier le plus proche, tandis que `inf` effectue un arrondi par défaut ; les fonctions `min` et `max` sont usuelles. La fonction `pos` (respectivement `pos_ou_nul`) renvoie 1 si son argument est strictement positif (respectivement positif ou nul), et 0 sinon. Les fonctions `present` et `null` sont utilisées pour discriminer la valeur `indéfini` des valeurs plus conventionnelles ; leur comportement est formellement défini dans la figure 6.

3.3 Typage du langage μM

Le typage du langage est défini dans la figure 4. Le but de ce système de type est d'assurer dans les programmes valides une distinction entre les variables scalaires et les variables de tableaux, ainsi que de séparer les types booléens et flottants.

Typage des expressions L'ensemble des variables est noté $\mathcal{V}$. Les types des valeurs sont $T = \{\texttt{booléen}, \texttt{flottant}\}$, tandis que les variables peuvent avoir un type scalaire simple ou un type tableau (ayant un contenu de type uniforme $\tau \in T$) $\Theta = \{\texttt{scalaire}[\tau], \texttt{tableau}[\tau]\}$. Le contexte de typage Γ est une fonction partielle $\Gamma : \mathcal{V} \rightharpoonup \Theta$ mémorisant le type des variables. Les expressions arithmétiques, logiques ou de comparaison sont considérées comme des appels de fonctions afin d'alléger la présentation. Le type des fonctions est constant et défini dans un environnement global Δ (présenté dans la figure 4). La majorité des fonctions admettent

⟨*programme*⟩ ::= ⟨*commande*⟩ | ⟨*commande*⟩ `;` ⟨*programme*⟩

⟨*commande*⟩ ::= ⟨*var*⟩ `:=` ⟨*expr*⟩
| ⟨*var*⟩ `[ X ;` ⟨*entier*⟩ `] :=` ⟨*expr*⟩
| `si` ⟨*expr*⟩ `alors` ⟨*erreur*⟩

⟨*expr*⟩ ::= ⟨*var*⟩ | `X` | ⟨*valeur*⟩
| ⟨*expr*⟩ ⟨*compop*⟩ ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩ | ⟨*unop*⟩ ⟨*expr*⟩
| `si` ⟨*expr*⟩ `alors` ⟨*expr*⟩ `sinon` ⟨*expr*⟩
| ⟨*func*⟩ `(` ⟨*expr*⟩, . . . , ⟨*expr*⟩ `)` | ⟨*var*⟩ `[` ⟨*expr*⟩ `]`

⟨*valeur*⟩ ::= `indéfini` | ⟨*défini*⟩

⟨*défini*⟩ ::= ⟨*booléen*⟩ | ⟨*flottant*⟩

⟨*compop*⟩ ::= `<=` | `<` | `>` | `>=` | `==` | `!=`

⟨*binop*⟩ ::= ⟨*arithop*⟩ | ⟨*logicop*⟩

⟨*arithop*⟩ ::= `+` | `-` | `*` | `/`

⟨*logicop*⟩ ::= `&&` | `||`

⟨*unop*⟩ ::= `-` | `~`

⟨*func*⟩ ::= `arr` | `inf` | `present` | `null`
| `max` | `min` | `abs` | `pos` | `pos_ou_nul`

FIGURE 2 – Syntaxe du langage μM

```
VAR0 = si (VAR1 > 0.0) alors - arr(VAR2 * 23 / 100) sinon 0.0;
TABLEAU[X; 10] = (3 * X * X + 2 * X + 1) * (1 - present(VAR1));
```

FIGURE 3 – Exemple de programme μM

un type spécifique en entrée et en sortie, à l'exception des fonctions `null` et `present`, qui sont polymorphes en entrée.

A l'exception de T-UNDEF et de T-VAR-UNDEF, les règles de typage sont classiques : les valeurs booléennes sont des booléens, et de même pour les flottants. La règle T-UNDEF reflète la grande permissivité du langage lors de l'utilisation des `indéfinis`, puisque ceux-ci sont considérés comme des booléens ou des flottants. Le type d'une variable x est τ si x est déclaré dans le contexte de typage comme étant un scalaire de type τ. De manière similaire à T-UNDEF, T-VAR-UNDEF correspond à un comportement particulier du langage M : une variable qui n'est pas définie est évaluée en `indéfini`, qui a donc n'importe quel type. Le domaine de Γ, dom Γ, est l'ensemble des variables définies dans le contexte Γ. La garde d'une conditionnelle doit être booléenne, et les expressions dans les branches de même type. L'accès dans un tableau est valide (pour le typage) dès lors que l'index est flottant. Un appel de fonction est bien typé si l'arité est correcte et si tous les arguments sont du type spécifié par une signature dans l'environnement global de typage des fonctions Δ.

Typage des commandes et programmes Le typage des commandes transforme le contexte de typage au fil des assignations. La notation $\Gamma[x \mapsto \theta]$ est le contexte Γ étendu avec l'annotation « x a le type θ ». Dans le cas d'une assignation scalaire x `:=` e, le contexte de typage est étendu d'une association entre x et `scalaire`$[\tau]$, où τ est le type de e. Dans le cas d'une assignation de tableau, $x[\texttt{X}, n]$ `:=` e , le contexte de typage est étendu pour lier x avec `tableau`$[\tau]$, où τ est le type de e dans l'environnement étendu où la variable générique d'index `X` est de type scalaire

Définition de l'environnement global de fonction Δ :

$\Delta(\texttt{arr}) = \Delta(\texttt{inf}) = \Delta(\texttt{abs}) = \{\texttt{flottant} \to \texttt{flottant}\}$

$\Delta(\texttt{pos}) = \Delta(\texttt{pos\_ou\_nul}) = \{\texttt{flottant} \to \texttt{booléen}\}$

$\Delta(\texttt{null}) = \Delta(\texttt{present}) = \{\tau \to \texttt{booléen} \mid \tau \in \{\texttt{booléen}, \texttt{flottant}\}\}$

$\Delta(\texttt{min}) = \Delta(\texttt{max}) = \Delta(\langle \textit{arithop} \rangle) = \{\texttt{flottant} \times \texttt{flottant} \to \texttt{flottant}\}$

$\Delta(\langle \textit{logicop} \rangle) = \{\texttt{booléen} \times \texttt{booléen} \to \texttt{booléen}\}$

$\Delta(\langle \textit{compop} \rangle) = \{\texttt{flottant} \times \texttt{flottant} \to \texttt{booléen}\}$

Jugement : $\boxed{\Gamma \vdash e : \tau}$ (« Sous Γ, e est de type τ »)

T-BOOL
$$\frac{}{\Gamma \vdash \langle \textit{booléen} \rangle : \texttt{booléen}}$$

T-FLOAT
$$\frac{}{\Gamma \vdash \langle \textit{flottant} \rangle : \texttt{flottant}}$$

T-UNDEF
$$\frac{\tau \in \{\texttt{booléen}, \texttt{flottant}\}}{\Gamma \vdash \texttt{indéfini} : \tau}$$

T-VAR-UNDEF
$$\frac{x \notin \text{dom } \Gamma \qquad \tau \in \{\texttt{booléen}, \texttt{flottant}\}}{\Gamma \vdash x : \tau}$$

T-VAR
$$\frac{\Gamma(x) = \texttt{scalaire}[\tau]}{\Gamma \vdash x : \tau}$$

T-INDEX-UNDEF
$$\frac{x \notin \text{dom } \Gamma \qquad \Gamma \vdash e : \texttt{flottant} \qquad \tau \in \{\texttt{booléen}, \texttt{flottant}\}}{\Gamma \vdash x[e] : \tau}$$

T-INDEX
$$\frac{\Gamma(x) = \texttt{tableau}[\tau] \qquad \Gamma \vdash e : \texttt{flottant}}{\Gamma \vdash x[e] : \tau}$$

T-CONDITIONAL
$$\frac{\Gamma \vdash e_1 : \texttt{booléen} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{si } e_1 \texttt{ alors } e_2 \texttt{ sinon } e_3 : \tau}$$

T-FUNC
$$\frac{\tau_1 \times \ldots \times \tau_n \to \tau \in \Delta(f) \qquad \Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

Jugement : $\boxed{\Gamma \vdash \langle \textit{commande} \rangle \Rrightarrow \Gamma'}$ et $\boxed{\Gamma \vdash \langle \textit{programme} \rangle \Rrightarrow \Gamma'}$ (« P transforme Γ en Γ' »)

T-ASSIGN
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x \texttt{ := } e \Rrightarrow \Gamma[x \mapsto \texttt{scalaire}[\tau]]}$$

T-ASSIGN-TABLE
$$\frac{\Gamma[\texttt{X} \mapsto \texttt{scalaire}[\texttt{flottant}]] \vdash e : \tau}{\Gamma \vdash x[\texttt{X}, n] \texttt{ := } e \Rrightarrow \Gamma[x \mapsto \texttt{tableau}[\tau]]}$$

T-COND
$$\frac{\Gamma \vdash e : \texttt{booléen}}{\Gamma \vdash \texttt{si } e \texttt{ alors } \langle \textit{erreur} \rangle \Rrightarrow \Gamma}$$

T-SEQ
$$\frac{\Gamma_0 \vdash c \Rrightarrow \Gamma_1 \qquad \Gamma_1 \vdash P \Rrightarrow \Gamma_2}{\Gamma_0 \vdash c \texttt{ ; } P \Rrightarrow \Gamma_2}$$

FIGURE 4 – Typage des expressions, commandes et des programmes

flottant. Les dernières règles sont classiques : les assertions doivent être de type booléen, et un programme est typé séquentiellement.

3.4 Sémantique du langage μM

Sémantique des expressions De manière générale, le langage M transforme les erreurs à l'exécution (telles que les divisions par zéro et les accès invalides dans les tableaux) en la valeur `indéfini` ; cette valeur peut ensuite être silencieusement convertie en d'autres valeurs, comme

l'illustre le comportement des opérateurs usuels en figure 6.

L'environnement mémoire d'exécution, noté Ω dans la suite, est une fonction partielle des variables vers des valeurs scalaires (dénotées dans la suite par v) ou le contenu d'un tableau (noté dans la suite comme un n-uplet $(v_0, ..., v_{n-1})$). Les valeurs litérales sont évaluées directement en valeurs, et les variables sont définies par l'environnement mémoire Ω (ou, à defaut, en la valeur `indéfini`). La variable spéciale d'indice de tableau `X` est considérée comme une variable normale. Les conditionnelles sont évaluées usuellement si la garde est évaluée en une valeur booléenne. Dans le cas où la garde est évaluée en `indéfini`, la conditionnelle est évaluée en `indéfini`. Si l'indice d'un tableau est évalué en `indéfini`, ou si l'indice est non-entier ou en dehors des bornes de définition, l'accès est évalué en `indéfini`. L'accès à un tableau avec un indice valide est traité de manière classique, en consultant l'environnement Ω. Le comportement des fonctions est défini en figure 6, avec en particulier la spécification du comportement de la valeur `indéfini` pour chaque opérateur, qui est une des spécificités de M (les opérateurs unaires se comportent comme les opérateurs binaires similaires, et ils sont donc omis).

Sémantique des commandes et programmes L'environnement mémoire pour les commandes et les programmes est celui des expressions étendu pour supporter `erreur`, qui est utilisé lorsqu'une assertion est invalide. La règle D-ERROR stipule que si l'environnement `erreur` est rencontré, celui-ci est propagé tout au long du programme. L'environnement étendu est noté Ω_c, il a pour valeur un environnement d'expression Ω ou `erreur`. Les assignations de variables scalaires transforment l'environnement, de manière similaire à la règle de typage de l'assignation. Les assertions pouvant lever des erreurs sont exécutées de la manière suivante : la garde est évaluée en une valeur ; si elle vaut `vrai`, l'erreur est levée. Si elle vaut `faux` ou `indéfini`, le programme continue son exécution. L'assignation d'un tableau $x[\texttt{X}, n]$ `:=` e est exécutée en évaluant e n fois, dans un environnement Ω étendu par la valeur de `X` correspondant à l'indice calculé. La séquence entre une commande et un programme est exécutée de manière usuelle.

3.5 Sûreté du typage

Le langage étant à la fois simple et permissif, nous arrivons à prouver que les programmes bien typés s'exécutent sans erreur (excepté les erreurs d'assertions). Pour définir la sûreté du typage, il faut d'abord s'assurer de la cohérence entre les environnements Γ et Ω :

Définition ($\Gamma \rhd \Omega$)**.** On dit que le contexte d'évaluation Ω est correctement représenté par le contexte de typage Γ, et l'on note $\Gamma \rhd \Omega$ si :

— Les types scalaires sont représentés par des valeurs du bon type :

$$\forall x \in \mathcal{V}, \tau \in T, \Gamma(x) = \texttt{scalaire}[\tau] \implies \exists v \in \langle \textit{valeur} \rangle, \Omega(x) = v \ \wedge \ \varnothing \vdash \text{Val } v : \tau$$

— Les types tableaux sont représentés par des tableaux ayant le bon type :

$$\forall x \in \mathcal{V}, \tau \in T, \Gamma(x) = \texttt{tableau}[\tau] \implies \exists n \in \mathbb{N}, \exists (v_0, \cdots, v_{n-1}) \in \langle \textit{valeur} \rangle^n,$$
$$\Omega(x) = (v_0, \cdots, v_{n-1}) \ \wedge \ \forall i \in [\![0; n-1]\!], \varnothing \vdash \text{Val } v_i : \tau$$

— Les variables non définies dans Γ ne le sont pas dans Ω non plus :

$$\forall x \in \mathcal{V}, x \notin \text{dom } \Gamma \implies x \notin \text{dom } \Omega$$

Théorème (Sûreté du typage des expressions)**.** Si $\Gamma \rhd \Omega$ et $\Gamma \vdash e : \tau$, alors il existe une valeur v telle que $\Gamma \vdash e \Downarrow v$ et $\varnothing \vdash \text{Val } v : \tau$.

Jugement : $\boxed{\Omega \vdash e \Downarrow v}$ (« sous Ω, e s'évalue vers v »)

$$\text{D-VALUE}\ \frac{v \in \langle \textit{valeur} \rangle}{\Omega \vdash v \Downarrow v} \qquad \text{D-VAR}\ \frac{\Omega(x) = v}{\Omega \vdash x \Downarrow v} \qquad \text{D-VAR-UNDEF}\ \frac{x \notin \text{dom}\ \Omega}{\Omega \vdash x \Downarrow \texttt{indéfini}}$$

$$\text{D-COND-TRUE}\ \frac{\Omega \vdash e_1 \Downarrow \texttt{vrai} \qquad \Omega \vdash e_2 \Downarrow v_2}{\Omega \vdash \texttt{si}\ e_1\ \texttt{alors}\ e_2\ \texttt{sinon}\ e_3 \Downarrow v_2}$$

$$\text{D-COND-FALSE}\ \frac{\Omega \vdash e_1 \Downarrow \texttt{faux} \qquad \Omega \vdash e_3 \Downarrow v_3}{\Omega \vdash \texttt{si}\ e_1\ \texttt{alors}\ e_2\ \texttt{sinon}\ e_3 \Downarrow v_3} \qquad \text{D-COND-UNDEF}\ \frac{\Omega \vdash e_1 \Downarrow \texttt{indéfini}}{\Omega \vdash \texttt{si}\ e_1\ \texttt{alors}\ e_2\ \texttt{sinon}\ e_3 \Downarrow \texttt{indéfini}}$$

$$\text{D-TAB-UNDEF}\ \frac{x \notin \text{dom}\ \Omega}{\Omega \vdash x[e] \Downarrow \texttt{indéfini}}$$

$$\text{D-INDEX}\ \frac{\Omega(x) = (v_0, \ldots, v_{n-1}) \qquad \Omega \vdash e \Downarrow r \qquad r \geqslant 0 \qquad r < n \qquad r' = \text{truncate}_{\mathbb{T}}(r)}{\Omega \vdash x[e] \Downarrow v_{r'}} \qquad \text{D-INDEX-UNDEF}\ \frac{\Omega \vdash e \Downarrow \texttt{indéfini}}{\Omega \vdash x[e] \Downarrow \texttt{indéfini}}$$

$$\text{D-INDEX-NEG}\ \frac{\Omega \vdash e \Downarrow r \qquad r < 0}{\Omega \vdash x[e] \Downarrow 0} \qquad \text{D-INDEX-OUTSIDE}\ \frac{\Omega \vdash e \Downarrow r \qquad r \geqslant n}{\Omega \vdash x[e] \Downarrow \texttt{indéfini}}$$

$$\text{D-FUNC}\ \frac{\Omega \vdash e_1 \Downarrow v_1 \qquad \cdots \qquad \Omega \vdash e_n \Downarrow v_n}{\Omega \vdash f(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)}$$

Jugement : $\boxed{\Omega_c \vdash c \Rrightarrow \Omega_c'}$ et $\boxed{\Omega_c \vdash P \Rrightarrow \Omega_c'}$ (« sous Ω_c, P produit Ω_c' »)

$$\text{D-ASSIGN}\ \frac{\Omega_c \neq \texttt{erreur} \qquad \Omega_c \vdash e \Downarrow v}{\Omega_c \vdash x\ \texttt{:=}\ e \Rrightarrow \Omega_c[x \mapsto v]} \qquad \text{D-ASSERT-OTHER}\ \frac{\Omega_c \neq \texttt{erreur} \qquad \Omega_c \vdash e \Downarrow v \qquad v \in \{\texttt{faux}, \texttt{indéfini}\}}{\Omega_c \vdash \texttt{si}\ e\ \texttt{alors}\ \langle \textit{erreur} \rangle \Rrightarrow \Omega_c}$$

$$\text{D-ASSERT-TRUE}\ \frac{\Omega_c \neq \texttt{erreur} \qquad \Omega_c \vdash e \Downarrow \texttt{vrai}}{\Omega_c \vdash \texttt{si}\ e\ \texttt{alors}\ \langle \textit{erreur} \rangle \Rrightarrow \texttt{erreur}} \qquad \text{D-ERROR}\ \frac{}{\texttt{erreur} \vdash c \Rrightarrow \texttt{erreur}} \qquad \text{D-SEQ}\ \frac{\Omega_{c,0} \vdash c \Rrightarrow \Omega_{c,1} \qquad \Omega_{c,1} \vdash P \Rrightarrow \Omega_{c,2}}{\Omega_{c,0} \vdash c\ ;\ P \Rrightarrow \Omega_{c,2}}$$

$$\text{D-ASSIGN-TABLE}\ \frac{\Omega_c \neq \texttt{erreur} \qquad \Omega_c[\texttt{X} \mapsto 0] \vdash e \Downarrow v_0 \qquad \cdots \qquad \Omega_c[\texttt{X} \mapsto n-1] \vdash e \Downarrow v_{n-1}}{\Omega_c \vdash x[\texttt{X}, n]\ \texttt{:=}\ e \Rrightarrow \Omega_c[x \mapsto (v_0, \ldots, v_{n-1})]}$$

FIGURE 5 – Évaluation des expressions, des commandes et des programmes

Afin de gérer les erreurs levées par les conditions de vérification, la relation $\triangleright$ est étendue aux environnements d'exécution des commandes de la façon suivante :

$$\Gamma \triangleright_c \Omega_c \iff \Omega_c = \texttt{erreur} \vee \Gamma \triangleright \Omega_c$$

Le théorème de sûreté du typage des commandes stipule qu'une commande bien typée dans des contextes en relation (au sens de $\triangleright_c$) sera correctement exécutée, et que les contextes en sortie seront aussi en relation.

Théorème (Sûreté du typage des commandes)**.** Si $\Gamma \vdash c \Rrightarrow \Gamma'$ et $\Gamma \triangleright_c \Omega_c$, alors il existe Ω_c' tel que $\Omega_c \vdash c \Rrightarrow \Omega_c'$ et $\Gamma' \triangleright_c \Omega_c'$.

3.6 Formalisation en Coq du langage μM

Afin de spécifier de manière précise la sémantique de μM et d'avoir une plus grande confiance en nos résultats, nous l'avons formalisée en Coq. Les jugements de typage sont définis comme des prédicats inductifs, tandis que la sémantique est définie de manière exécutable. Les théorèmes

$f_1 \odot f_2, \odot \in \{+, -\}$	`indéfini`	$f_2 \in \mathbb{F}$
`indéfini`	`indéfini`	$0 \odot f_2$
$f_1 \in \mathbb{F}$	$f_1 \odot 0$	$f_1 \odot f_2$

$f_1 \odot f_2, \odot \in \{\times, \div\}$	`indéfini`	$f_2 \in \mathbb{F}, f_2 \neq 0$
`indéfini`	`indéfini`	`indéfini`
f_1	`indéfini`	$f_1 \odot f_2$

$b_1 \langle logicop \rangle b_2$	`indéfini`	$b_2 \in \mathbb{B}$
`indéfini`	`indéfini`	`indéfini`
$b_1 \in \mathbb{B}$	`indéfini`	$b_1 \langle logicop \rangle b_2$

$f_1 \langle compop \rangle f_2$	`indéfini`	$f_2 \in \mathbb{F}$
`indéfini`	`indéfini`	`indéfini`
$f_1 \in \mathbb{F}$	`indéfini`	$f_1 \langle compop \rangle f_2$

$f_1 \times 0 = 0$

$\texttt{present}(v) = (v \neq \texttt{indéfini})$

$\texttt{null}(\texttt{indéfini}) = \texttt{indéfini}$

$\texttt{null}(v \neq \texttt{indéfini}) = (v \neq 0)$

$\texttt{pos(x)} \equiv \texttt{x > 0}$

$\texttt{arr}(\texttt{indéfini}) = 0$

$\texttt{arr}(f \in \mathbb{F}) = \text{round}_{\mathbb{F}}(f)$

$f_1 \div 0 = \texttt{indéfini}$

$\texttt{abs(x)} \equiv \texttt{si x >= 0 alors x sinon -x}$

$\texttt{min(x, y)} \equiv \texttt{si x >= y alors y sinon x}$

$\texttt{max(x, y)} \equiv \texttt{si x >= y alors x sinon y}$

$\texttt{pos\_ou\_nul (x)} \equiv \texttt{x >= 0}$

$\texttt{inf}(\texttt{indéfini}) = 0$

$\texttt{inf}(f \in \mathbb{F}) = \text{truncate}_{\mathbb{F}}(f)$

FIGURE 6 – Comportement des fonctions

de sûreté énoncés précédemment ont été prouvés en Coq. Le développement fait actuellement 850 lignes, et est disponible publiquement [5].

3.7 Compilateur du langage M et évaluation

Implémentation Nous avons implémenté en OCaml un compilateur appelé MLANG, capable de lire le code source mis à disposition par la DGFiP, d'ordonnancer ses assignations et d'interpréter le programme résultant, ou de le traduire vers du code Python exécutable. Ce compilateur est disponible publiquement [6] sous license GPL. Le compilateur permet également de spécialiser le code et de lui appliquer des optimisations classiques (évaluation partielle, numérotage global des valeurs, élimination de code mort).

Évaluation Nous avons ensuite pu vérifier que notre implémentation respectait la sémantique de M en utilisant des tests fournis par la DGFiP. Chaque test spécifie les valeurs de certaines variables d'entrées, ainsi que des valeurs de variable à vérifier ; une seule contrainte entre variable et valeur est définie par ligne. 542 tests nous ont été fournis ; les tests font de 74 à 713 lignes, avec une médiane à 597 lignes. Nous passons actuellement 104 tests, ayant chacun entre 558 et 674 lignes. L'examen des tests ne passant pas nous a permis de mettre à jour des informations manquantes quand à l'exécution du code M non encore publié par la DGFiP. Plus précisément, certaines valeurs d'entrées nécessaires aux tests ne sont pas spécifiées dans les tests mais par une autre application. La validation de notre artefact sur le jeu complet de tests de la DGFiP est donc laissé en travail à venir.

4 Étude formelle de l'impôt

Grâce à la sémantique du langage M, le code utilisé par la DGFiP pour calculer l'impôt sur le revenu devient une formalisation de la portion algorithmique du code des impôts. Il est

5. `https://gitlab.inria.fr/verifisc/mlang/tree/master/formal_semantics`
6. `https://gitlab.inria.fr/verifisc/mlang/tree/master/src/mlang`

possible d'utiliser des techniques classiques de méthodes formelles afin d'analyser le code. De plus, le fait que le langage M ne soit pas Turing-complet nous simplifie la tâche.

Afin d'explorer les applications potentielles de notre formalisation sans attendre la validation complète à venir de Mlang, nous avons décidé de créer un deuxième artefact, indépendant du code M, dans lequel nous réimplémentons un cas simplifié de l'impôt sur le revenu, auquel nous avons ajouté le calcul de diverses allocations. Les résultats de cette section sont donc basés sur ce deuxième artefact.

4.1 Utilisation de solveurs SMT

À condition de modéliser un montant d'argent comme un nombre entier de centimes (ou un réel à précision fixe), il est possible d'encoder le programme M de calcul des impôts dans un fragment logique contenant uniquement l'arithmétique et la logique du premier ordre. Il est alors possible d'encoder le calcul de l'impôt dans un solveur SMT comme Z3 [3].

Nous avons testé empiriquement plusieurs encodages SMT pour le calcul arithmétique de l'impôt : l'arithmétique entière non-bornée (avec un nombre entier de centimes), l'arithmétique réelle non-bornée (1,23 € étant encodé comme 1.23) et l'arithmétique entière bornée (encodée comme opérations sur des vecteurs de bits). Cet encodage a été testé sur l'impôt sur les revenus de 2017, qui ne relève pas du fragment linéaire de l'arithmétique. En effet, au moins une provision du codes des impôts de l'époque, la réduction d'impôt prévue au (b) du 4 du I de l'article 197 du CGI [7] dépend du carré du revenu fiscal de référence du foyer.

Dans ces conditions, seul l'encodage à base de vecteurs de bits nous a permis empiriquement d'obtenir une réponse de manière systématique sans tomber dans l'incomplétude des procédures de décision de Z3. Un encodage plus performant serait probablement possible, d'autant plus que la réforme du barème de l'impôt sur le revenu de 2019 supprime la seule partie non-linéaire du calcul. Cependant, nous considérons ici uniquement l'encodage sur vecteurs de bits.

Dans ce cas, quelle taille de vecteur choisir ? Étant donné qu'un vecteur de bits représente un nombre entier de centimes, il nous faut estimer la valeur intermédiaire maximale atteinte lors du calcul de l'impôt afin d'éviter tout débordement arithmétique. Dans le fragment linéaire du calcul, cette valeur maximale est atteinte lorsque l'on veut prendre une fraction d'un revenu. Plus la fraction que l'on veut prendre est précise, plus la valeur intermédiaire atteinte est grande. Pour calculer $42,8\%$ d'un revenu X, il faut calculer $X \times 428/1000$. Dans notre code, nous nous limitons à une précision de l'ordre du dixième de pourcentage (donc un facteur multiplicatif de moins de 1000), suffisante pour approximer le calcul officiel à un ou deux euros près. La provision non-linéaire est encodée de manière spéciale sur un vecteur de bits deux fois plus grand pour éviter les débordements. Avec ces contraintes, si l'on veut pouvoir modéliser plus de 99% des cas avec un revenu annuel maximal de 105 000 euros par individu du foyer [8], alors un nombre de bits égal à 34 suffit. On peut choisir un nombre de bits plus élevé pour calculer l'impôt de plus hauts revenus mais ceci impact le temps de calcul de Z3.

Nous avons développé en utilisant l'interface Python de Z3 un prototype open-source [9] modélisant le calcul de l'impôt sur le revenu ainsi que diverses allocations pour les ménages. Ce prototype fait un certain nombre d'hypothèses qui restreignent l'espace possible des ménages considérés ; globalement, on considère des locataires n'ayant que des revenus salariaux. Cependant, toutes les combinaisons de situation matrimoniale (y compris le concubinage) et de distribution d'enfants à charge (là aussi le nombre maximal est borné, par défaut à 6) sont

7. `https://bofip.impots.gouv.fr/bofip/2495-PGP`
8. `https://www.insee.fr/fr/statistiques/fichier/3549487/REVPMEN18_D1_tres-hauts-revenus.pdf`, p.4
9. `https://gitlab.inria.fr/verifisc/verifisc-python`

Caractéristique	Valeur avant	Valeur après	Variation
Salaire mensuel net premier individu	**2 760,76 €**	**3 010,76 €**	+ 250,00 €
Salaire mensuel net deuxième individu	**0,00 €**	0,00 €	0,00 €
Revenus annuels des salaires du foyer	33 129,12 €	36 129,12 €	**+ 3 000,00 €**
Revenu fiscal de référence (annuel)	29 816,00 €	32 516,00 €	+ 2 700,00 €
Impôt sur le revenu (annuel)	3 147,00 €	3 957,00 €	+ 810,00 €
Prime d'activité (mensuelle)	110,00 €	0,00 €	− 110,00 €
Allocations familiales (mensuelles)	132,00 €	132,00 €	0,00 €
Allocation rentrée scolaire (annuelle)	806,00 €	0,00 €	− 806,00 €
Bourse collège (trimestrielle)	0,00 €	0,00 €	0,00 €
Bourse lycée (annuelle)	0,00 €	0,00 €	0,00 €
Allocations Personnalisée au Logement (mensuelle)	0,00 €	0,00 €	0,00 €
Net touché après redistribution	33 692,12 €	33 756,12 €	**+ 64,00 €**
Augmentation des revenus mensuels après redistribution			**5,33 €**
Taux marginal			**97,9 %**

FIGURE 7 – Couple en concubinage avec deux enfants de 15 et 17 ans scolarisés et à charge du deuxième individu (sans activité). Le ménage habite en zone II dans une location pour 897,75 €. On suppose que le premier individu du couple est augmenté de 250 € par mois. L'augmentation de l'impôt sur le revenu, ainsi que la perte de la prime d'activité et de l'allocation de rentrée scolaire, conduisent à ce que le couple touche uniquement 2,1 % de l'augmentation initiale.

prises en compte par le modèle. Z3 nous permet alors de répondre à des questions de la forme « Existe-t-il un ménage X tel que $P(X)$ », où $P(X)$ est un prédicat numérique dépendant des caractéristiques du ménage X et du montant des différents impôts et revenus calculés. Il est possible d'encoder dans P des questions impliquant l'évolution du ménage entre deux années, soit $P(X) = Q(X, f(X))$ où f décrit la transformation du ménage d'une année sur l'autre (en supposant pour notre prototype que la législation fiscale ne change pas) et Q étant notre prédicat numérique.

Aussi, en prenant pour f l'augmentation du salaire du premier individu du ménage de x euros par mois, et pour Q le fait que l'augmentation du revenu après redistribution (impôts et allocations) du ménage représente moins de y % des x euros d'augmentation, alors Z3 nous permet de répondre à la question de l'existence d'une taxation marginale plus élevée que $(100 - y)$ %. En effet, le taux marginal de taxation est défini comme le taux d'impôt prélevé sur une petite augmentation de revenus par rapport à une situation initiale. Plus précisément, nous considérons le taux marginal effectif de prélèvement (TMEP) car nous prenons en compte à la fois impôts et allocations.

La figure 7 présente une des réponses trouvées par Z3 à la recherche de taux marginaux élevés. La correction de l'encodage du calcul de l'impôt dans le prototype a été testée sur de nombreux exemples en comparant avec les simulateurs officiels de l'impôt sur le revenu et des aides sociales. Les montants sont corrects à 1-2 € près, la différence étant sûrement imputable à des divergences concernant l'arrondi des montants dans le calcul et aux approximations de fractions liées à l'encodage (décrites plus haut). Par contre, la correction des résultats négatifs retournés par Z3 (« il n'existe pas de foyer tel que ... ») est conditionnée à l'absence de bugs.

La question de la recherche de taux marginaux élevés est une question complexe, qui nécessite « d'inverser » la fonction de calcul de l'impôt, puisque l'on cherche des foyers fiscaux à partir d'un montant d'impôt payé. Cette inversion réalisée par le solveur SMT est une amélioration inédite par rapport aux autres outils d'analyse utilisés en économie pour évaluer les effets d'une

formule de calcul de l'impôt. En effet, les travaux récents évaluant ce TMEP [5] utilisent une dérivation partielle manuelle qui néglige les arrondis à l'euro près et une augmentation de revenus fixe pour tous les ménages à 3 % des revenus professionnels. Le calcul se fait sur un jeu de données correspondant à des ménages réels, mais qui couvre généralement les revenus sur une année et ne permettent pas de suivre l'évolution d'un ménage. De plus, l'accès à ces données est protégé par le secret fiscal. Pour cette raison, traditionnellement, seul l'Insee peut produire de telles analyses.

La possibilité d'explorer tout l'espace des ménages possibles permet également d'anticiper des situations indésirables résultant par exemple de l'interaction entre plusieurs dispositifs législatifs, comme l'illustre la figure 7. L'analyse économique des transferts sociaux aidée par SMT ouvre donc des pistes prometteuses qui dépassent le cadre de cet article. Cependant, la question du passage à l'échelle va se révéler cruciale : en effet, notre prototype prend déjà plusieurs heures et des dizaines de Go de mémoire vive afin de répondre aux requêtes portant sur l'impôt sur le revenu dans un cas simplifié. Nous avons donc besoin d'outils fiables pour guider le travail du solveur SMT et lui présenter des requêtes les plus simples possibles.

4.2 Travaux futurs : interprétation abstraite

Revenons maintenant au langage M et à la base de code existante maintenue par la DGFiP qui encode le calcul de l'impôt. Il est possible de traduire un programme M vers un encodage compatible avec un solveur SMT. Cependant, comme mentionné auparavant, cela produirait des requêtes trop volumineuses pour être traitées en temps raisonnable. Notre stratégie est donc de circonscrire le problème à une situation simplifiée par rapport au cas général : par exemple, le cas où le ménage est un couple marié qui ne possède que des revenus salariaux et des enfants sans aucun cas spécial. Dans ce cas simplifié, les optimisations classiques (élimination de code mort, numérotage global des valeurs, évaluation partielle), implémentées dans le compilateur Mlang, font passer le nombre de variables du programme d'environ 46 000 à 650. Mais même ainsi, le programme résultant est trop volumineux pour le solveur SMT. En effet, le code M contient des motifs récurrents comme celui de l'écrêtage de valeur par un seuil constant (expressions de la forme `VAR1 = min(VAR0, 20)`). Pour simplifier ces motifs, une interprétation abstraite utilisant un domaine numérique semble être tout indiquée. Nous envisageons donc dans le futur de connecter Mlang à l'outil d'interprétation abstraite MOPSA [7], et d'utiliser les informations fournies par l'analyse afin de simplifier le programme sous un ensemble d'hypothèses sur les données en entrée (revenus positifs, etc).

Nous espérons que l'interprétation abstraite, combinée à une division des requêtes en sous-problèmes spécialisés sur critères décidés par l'utilisateur (par exemple le cas célibataire, le cas marié et le cas concubinage), nous permette de produire des requêtes optimisées traitables rapidement par un solveur SMT avec un encodage des entiers en vecteurs de bits.

5 Travaux connexes

Diverses initiatives ont déjà été tentées afin de formaliser une petite partie d'une législation ou d'une réglementation. Certaines sont très anciennes comme celle de Allen [1] en 1956, qui affine sa méthode en 1978 [2] pour utiliser un langage de diagrammes. Kowalski et al. [14] utilisent Prolog en 1986 sur la détermination de la nationalité britannique. Plus récemment, Sarah Lawsky a mené deux études de cas [8, 9] portant sur des fragments du droit fiscal américain, et propose d'utiliser la logique par défaut [13] pour encoder la structure de la législation. La difficulté que rencontrent toutes ces initiatives provient de la traduction de la loi en algorithme :

seuls certains fragments de la législation s'y prêtent, et même dans ce cas le texte de loi n'est pas écrit dans un style programmatique, ce qui rend la traduction ardue.

En ce qui concerne les contrats, Hvitved propose une spécification de contrats multipartites qu'il formalise [6] dans un DSL. Il existe un très grand nombre de travaux portant sur ce sujet, originant des communautés de la logique formelle, du droit, de l'intelligence artificielle (avant l'apparition de l'apprentissage statistique), et plus récemment des systèmes distribués (blockchain et « smart contracts »).

Ce domaine de recherche est également exploré par des sociétés privées. En Espagne, une initiative menée par la société Formal Vindications a formalisé les conséquences d'une réglementation européenne sur la conduite [4] pour mieux en montrer les incohérences. Au Royaume-Uni, la société Aesthetic Integration formalise le fonctionnement des plate-formes des marchés financiers et prouve leur conformité par rapport aux réglementations financières [11]. Ces travaux constituent sans doute l'initiative la plus aboutie de formalisation de texte législatif, qui se base sur un langage et une suite logicielle appelée Imandra combinant plusieurs techniques issues de la recherche en méthodes formelles.

Parallèlement à ces efforts de formalisation, un autre axe de recherche est l'implémentation de textes législatifs dans une base de données de règles couplée à un moteur d'exécution. En Nouvelle-Zélande, l'outil Datalex [10] est utilisé pour offrir une interface de type « chatbot » à des utilisateurs qui désireraient comparer une situation à la législation existante. Cette approche est à rapprocher de la catégorie des « Business Rules Management Systems » comme Drools [12], qui n'a pas pour objectif une formalisation complète mais plutôt le développement d'un outil pratique. On peut également citer l'initiative OpenFisca [10] menée par la Direction Interministérielle du Numérique et des Systèmes d'Information et de Communication, qui propose une implémentation en Python du système socio-fiscal français.

À notre connaissance, notre travail est inédit quant à l'approche systématique et la taille de la portion de législation formalisée. En effet, la France est le seul pays ayant publié en open-source le code calculant l'impôt sur le revenu de ses concitoyens. La plupart des travaux existants se focalisent sur des contrats, plus limités dans leur portée. La formalisation de la législation existante est moins intéressante à cause de la taille imposante des textes de lois et de leur inadaptation au raisonnement logique (incohérences, imprécisions). L'implémentation en M du code des impôts, maintenue depuis trois décennies par la DGFiP au prix d'un travail important, a été la brique de base indispensable à notre étude.

6 Conclusion

L'application des méthodes formelles à l'artefact logiciel publié par la DGFiP ouvre des pistes très prometteuses quant à l'analyse de l'impact de l'impôt sur le revenu. Le langage M, bien qu'ancien, relève d'un choix éclairé d'architecture logicielle dans le contexte de l'administration publique des années 90. La centralisation de l'encodage du code des impôts à l'aide d'un DSL dans une implémentation unique et mutualisée entre les applications clientes nous a permis de concentrer efficacement notre travail de formalisation. La formalisation de M présentée dans cet article, permet également la compilation de M vers virtuellement n'importe quel autre langage, de manière à fournir une implémentation clés en main du code des impôts à toutes les applications qui en auraient besoin : analyses macroéconomiques, simulateurs en ligne, etc.

Néanmoins, la correction du code M par rapport au code des impôts reste un sujet critique. En effet, un bug d'implémentation est potentiellement source de contentieux entre l'État et le

10. `https://openfisca.org`

particulier lésé par un calcul de l'impôt erroné. Le langage M n'a pas une structure permettant de se convaincre ou de vérifier modulairement cette correction fonctionnelle. La prochaine étape de ce travail est donc de créer un langage inspiré de M, mais qui serait utilisé pour annoter la loi dans un style de programmation littérale, article par article. Ces annotations tiendraient lieu de spécification formelle et pourraient aussi guider le législateur, à l'aide d'un outillage adapté mettant en valeur grâce à de l'inférence des incohérences ou des cas non-spécifiés.

Références

[1] Layman E Allen. Symbolic logic : A razor-edged tool for drafting and interpreting legal documents. *Yale LJ*, 66 :833, 1956.

[2] Layman E. Allen and C. Rudy Engholm. Normalized legal drafting and the query method. *Journal of Legal Education*, 29(4) :380–412, 1978.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] D Fernández Duque, M González Bedmar, D Sousa, Joosten, J.J, and G. Errezil Alberdi. To drive or not to drive : A formal analysis of requirements (51) and (52) from regulation (eu) 2016/799. In *Personalidades jurídicas difusas y artificiales*, pages 159–171. TransJus Working Papers Publication - Edición Especial, 4 2019.

[5] Juliette Fourcot and Michaël Sicsic. Les taux marginaux effectifs de prélèvement pour les personnes en emploi en france en 2014, 2017.

[6] Tom Hvitved. *Contract formalisation and modular implementation of domain-specific languages.* PhD thesis, Citeseer, 2011.

[7] M. Journault, A. Miné, M. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyszer. In *Proc. of VSTTE19*, pages 1–17, Jul. 2019.

[8] Sarah B. Lawsky. Formalizing the Code. *Tax Law Review*, 70(377), 2017.

[9] Sarah B. Lawsky. A Logic for Statutes. *Florida Tax Review*, 2018.

[10] Andrew Mowbray, Philip Chung, and Graham Greenleaf. Utilising ai in the legal assistance sector – testing a role for legal information institutes. In *LegalAIIA*, 2019.

[11] Grant Olney Passmore and Denis Ignatovich. Formal verification of financial algorithms. In *CADE*, 2017.

[12] Mark Proctor. Drools : A rule engine for complex event processing. In *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, AGTIVE'11, pages 2–2, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] R. Reiter. Readings in nonmonotonic reasoning. chapter A Logic for Default Reasoning, pages 68–93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[14] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The british nationality act as a logic program. *Commun. ACM*, 29(5) :370–386, May 1986.

[15] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, Oct 1971.

Des promesses, des actions, par flots, en OCaml

Simon Archipoff[1], David Janin[1], and Bernard Serpette[2]

[1] UMR CNRS LaBRI, Bordeaux INP,
Université de Bordeaux
Simon.Archipoff | David.Janin@labri.fr
[2] Inria Bordeaux Sud-Ouest
Bernard.Serpette@inria.fr

Résumé

Nous nous intéressons aux traitements de flux audio synchrones et aux contrôles de ces traitements par des flux de commandes asynchrones : un exemple archétypal de problème de programmation globalement asynchrone et localement synchrone (GALS). Pour faire cela, un modèle qui s'impose tout à la fois par son efficacité et son élégance se situe à la croisée de plusieurs concepts clés de la programmation fonctionnelle : les actions monadiques, les promesses, et les flots récursifs d'actions ou de promesses. Suite à une expérimentation de cette modélisation en Haskell, l'objet de cet article est d'en faire une expérimentation en OCaml. De façon quelque peu inattendue, les deux approches se complètent et nous conduisent finalement à une axiomatisation originale de la notion de promesse. De plus, étendu aux flots récursifs d'actions, il apparaît que les promesses de flots d'actions peuvent être vue comme des flots de promesses. Une mise en œuvre de ces concepts est proposée en s'appuyant sur les extensions d'OCaml par threads Posix ou par threads légers (lwt).

1 Introduction

Poursuivant notre étude de la modélisation de systèmes multimédia interactifs et temporisés [1, 2] (musique, son, animation, etc...) pour leur programmation à l'aide de languages fonctionnels typés génériques tels qu'OCaml ou Haskell, nous nous intéressons aujourd'hui aux traitements de flux audio synchrones et aux contrôles de ces traitements par des flux de commandes asynchrones : un exemple archetype de problème de programmation globalement asynchrone et localement synchrone (GALS).

Actions monadiques. Depuis les travaux de Moggi [13] et de Wadler [14], on sait qu'il est possible de modéliser les effets de bords en programmation fonctionnelle pure grâce à la notion d'actions monadiques.

En première approche, une action monadique peut être vue comme une fonction qui prend en entrée un « état du monde » et produit une valeur tout en modifiant cet « état du monde ». Cette modification de l' « état du monde » modélise l'effet de bord provoqué par l'application de cette action.

Sans forcément être identifiée comme telle, cette approche est omniprésente dans l'étude de la sémantique des langages de programmation. Par exemple, la sémantique d'une affectation `x := e` dans un langage impératif peut être vue comme l'action monadique qui prend en entrée l'état mémoire d'un programme en cours d'exécution, un état dans lequel la valeur de la variable `x` est définie, évalue l'expression `e` et modifie la valeur de la variable `x` en conséquence.

Avantage de cette approche, les effets de bord sont limités aux états de la monade considérée. On peut alors analyser les propriétés des programmes en raisonnant sur l'action qu'ils ont sur les états de la monade. La logique de Hoare avec pré et post-conditions sur les programmes impératifs en est un exemple.

Flots monadiques. Nous nous proposons ici d'étudier une application de cette programmation monadique à la programmation par flots de données.

Typiquement, en programmation par flots de données, l'accès à une nouvelle entrée, tout comme la production d'une nouvelle sortie, est un effet de bord. En suivant l'approche de Moggi et Wadler, il est donc naturel de modéliser ce traitement d'un nouvel élément d'un flot de données par une action monadique. De plus, le traitement de cet élement ouvre aussi, pourrait-on dire, l'accès à la suite de ce flot. Il apparait alors qu'un flot de données peut être modélisé par une action monadique qui, lorsqu'elle est évaluée, produit, selon le cas :

(1) un marqueur de fin dans le cas où le flot est terminé,

(2) une paire constituée de la valeur de l'élément courant et de la continuation de ce flot de données dans le cas contraire.

Autrement dit, ces flots, que nous appellerons flots monadiques, apparaissent comme des listes explicitement paresseuses, dont l'accès est systématiquement gardé par une action monadique.

Bien entendu, on le devine ici, nous ne modélisons que des flots discrets ou, pour être plus précis, des flots localement finis. Plus précisément, un flot de données est dit *localement fini* lorsque le nombre de valeurs produites par ce flot sur tout intervalle de temps de durée finie est nécessairement fini. Cette définition exclut donc les flots de type « Zenon [1] » qui ne sont pas localement fini puisqu'ils présentent des points temporels d'accumulation d'une infinité de valeurs.

Fonctions de flots. L'un des points clés de notre approche pour la programmation par flots de données est qu'une même structure de données, la notion de flot monadique, permet de représenter tout à la fois les flots d'entrées et les flots de sorties. En effet, un flot monadique apparait comme :

(1) un flot d'entrée lorsqu'on *exécute* (récursivement) les actions (imbriquées) qui le définissent pour *lire* les valeurs qu'il porte,

(2) un flot de sortie lorsqu'on *produit* (récursivement) les actions (imbriquées) qui le définissent pour *écrire* les valeurs qu'il porte.

Dans le codage d'une fonction de flots, la correspondance entre l'exécution des actions d'entrée et la production des actions de sortie définit alors la synchronicité entre les entrées et les sorties. La programmation monadique offre un contrôle total de cette synchronicité. En particulier, lorsque les entrées sont cadencées au même rythme que les sorties, on parlera de fonctions synchrones, et notre approche permet, comme nous le verrons, de programmer toutes les fonctions synchrones définissables à l'aide d'une machine de Mealy.

Et des promesses. Pour une application réelle de notre approche aux traitements des flots de données, il ne suffit pas de pouvoir définir toutes ces fonctions synchrones. Le contrôle de ces fonctions, souvent cadencés à des fréquences moins élevées que les flots traités, doit aussi pouvoir être décrits et mis en œuvre. Les mécanismes de concurrence asynchrone offerts par la notion de promesses, initialement appelées *future values* [4], permettent cela [3, 12, 11].

Plus encore, la notion de promesse va aussi nous permettre de partager des flots de données sans en partager les effets de bord. Par exemple, un flots d'entrée typique que nous définirons ici, est le flot des entrées standards. Sa lecture fait appel, de façon répétitive, à une action monadique `getChar` qui renvoie (de façon bloquante) le caractère suivant entré sur le clavier. Le partage de ce flot monadique parmi plusieurs processus indépendants ne conduit pas à

1. voir https://fr.wikipedia.org/wiki/Paradoxes_de_Z%C3%A9non

dupliquer ces entrées mais, au contraire, à les distribuer, chaque processus utilisateur de ce flux exécutant ses propres instances de l'action `getChar`.

Le partage de la même suite de caractères d'entrée parmi plusieurs processus peut être réalisé par une promesse de flots, une promesse qui, comme nous le verrons, peut se définir simplement et efficacement comme un flot de promesses.

2 Actions monadiques

En Haskell [5], langage de programmation paresseux et donc sans effets de bord, la programmation par monades est nécessaire et, d'une certaine façon, omniprésente. Le codage des flots monadiques est donc relativement aisé. Le lecteur interessé pourra consulter un exemple de ce codage [7] utilisé pour du traitement et du controle audio temps réel.

En OCaml [10], langage de programmation strict avec effets de bord, la programmation par monades est d'un usage beaucoup plus confidentiel parce que largement inutile. La plupart des instructions s'exécutent dans une monade IO implicite mais omniprésente. Nous proposons ici un codage générique de ces monades et une instanciation particulière qui explicite et mime en quelque sorte la monade IO d'Haskell.

Un aspect important de la programmation monadique est que les actions dites monadiques qu'on y manipule sont des valeurs comme les autres. Sauf à être passées en argument d'une fonction d'`exec`, elles ne sont pas exécutées. Au contraire, la programmation monadique consiste à créer puis à combiner des actions les unes avec les autres, parfois en prenant d'autres actions en paramètres, produisant ainsi des programmes toujours plus complexes qui seront, in fine, exécutés dans l'ordre prescrit, lors de l'exécution du programme principal.

Autrement dit, les actions monadiques forment un type de données qui est, par essence, paresseux. En OCaml, le codage le plus simple de la paresse consiste à définir des *glaçons*[2], c'est à dire une fonction de type `unit -> 'a`.

Les actions monadiques seront alors définies comme des glaçons encapsulés par un constructeur de type afin de les distinguer, par typage, des fonctions de type `unit -> 'a` que nous pourrions utiliser par ailleurs. Dans ce codage, l'état du monde évoqué en introduction reste implicitement représenté par l'état mémoire du programme OCaml. L'unique valeur de type `unit`, passée en paramètre d'un glaçon, ne sert qu'à déclencher son évaluation.

2.1 Définition générique des monades : module `Monad`

En OCaml, la signature d'une monade peut être définie par :

```
module type MONAD_CORE =
  sig
    type 'a m
    val return : 'a -> 'a m
    val bind   : 'a m -> ('a -> 'b m) -> 'b m
  end
```

avec le constructeur de type `m` pour les actions monadiques, la fonction `return` qui permet de

2. En anglais, ces glaçons s'appellent des *thunks*, un mot dont nous avons échoué à trouver une traduction littérale adéquate. Bien entendu, à la différence des *thunks* utilisés pour coder une évaluation paresseuse, nous ne sauvegardons pas la valeur retournée, deux executions distinctes d'une même action, telle `getChar`, pouvant retourner des valeurs différences.

créer une action monadique, et la fonction `bind` qui permet de combiner deux actions monadiques, la seconde action prenant en paramètre le résultat de la première. Ces fonctions doivent satisfaire les équations suivantes :

$$\texttt{bind (return a) f} \equiv \texttt{f a} \quad (1)$$

$$\texttt{bind m return} \equiv \texttt{m} \quad (2)$$

$$\texttt{bind (bind m f) g} \equiv \texttt{bind m (fun a -> bind (f a) g)} \quad (3)$$

pour toute valeur `a : a`, toute action monadique `m : 'a m` et toutes fonctions `f : 'a -> 'b m` et `g : 'b -> 'c m`. Les équations (1) et (2) assurent que, en un certain sens, la fonction `return` agit comme un neutre à gauche et à droite pour la fonction `bind`, l'équation (3) assurant que ce `bind` est associatif.

Les primitives d'une monade, définies par un module de type `MONAD_CORE`, sont systématiquement étendues par les fonctions suivantes :

```
module type MONAD =
  sig
    include MONAD_CORE
    val (>>=) : 'a m -> ('a -> 'b m) -> 'b m
    val (>>) : 'a m -> 'b m -> 'b m
    val fmap : ('a -> 'b) -> 'a m -> 'b m
  end
```

définies par :

```
module Monad (M: MONAD_CORE) : MONAD
       with type 'a m = 'a M.m
  =
  struct
    type 'a m = 'a M.m
    let return = M.return
    let bind = M.bind
    let (>>=) = bind
    let (>>) m1 m2 = bind m1 (fun _ -> m2)
    let fmap f m = bind m (fun x -> return ( f x))
  end
```

L'opérateur `(>>=)` n'est qu'une redéfinition infixe de la fonction `bind`. La fonction `fmap` explicite le fait que le constructeur de type `m` dans une monade peut être vu comme un foncteur envoyant l'identité sur l'identité et la composition de deux fonctions sur la composition de leurs images. Grâce aux équations (1)–(3), on peut en effet prouver que :

$$\texttt{fmap id} \equiv \texttt{id} \quad (4)$$

$$\texttt{fmap (g} \circ \texttt{f)} \equiv \texttt{(fmap g)} \circ \texttt{(fmap f)} \quad (5)$$

pour `id = fun x -> x` et toutes fonctions `f : 'a -> 'b` et `g : ' b -> 'c`.

D'un point de vue sémantique, nous dirons que deux actions sont équivalentes lorsqu'elles ont le même comportement dans tout contexte d'utilisation, effets de bord inclus. Autrement dit, deux actions sont équivalentes lorsqu'elles agissent de la même façon sur l'état de la monade et produisent des valeurs équivalentes dans tout contexte d'utilisation.

2.2 Une instance de monade : la monade `IO`

Un exemple simple de monade, mimant en quelque sorte la monade IO d'Haskell, l'état du monde étant implicitement définit par le runtime d'OCaml, est obtenu en définissant `'a io`, le type des actions monadiques retournant une valeur de type `'a`, comme une encapsulation du type des glaçons `unit -> 'a`. Plus précisément, on pose :

```
type 'a io = IO of (unit -> 'a)
module IO_CORE : MONAD_CORE
     with type 'a m = 'a io
  =
  struct
    type 'a m = 'a io
    let return a = IO (fun () -> a)
    let bind (IO g) f
      = IO (fun () -> let a = g () in let IO m = f a in m ())
  end
```

On étend alors la monade résultante d'une fonction `run` qui nous permet d'exécuter, en temps utile, une action IO, en posant :

```
module type MONAD_IO =
  sig
    include MONAD
    val run : 'a m -> 'a
  end

module IO : MONAD_IO with type 'a m = 'a io =
  struct
    include Monad (IO_CORE)
    let run (IO s) = s ()
  end
```

Il est important de se souvenir que les actions monadiques ne sont pas exécutées, sauf à être passées en argument de la function `run` ci-dessus.

Deux exemples classiques d'actions dans la monade IO sont les suivantes :

```
let getChar : 'char io
  = IO (fun () -> input_char stdin)

let printChar c : unit io
  = IO (fun () -> print_char c)
```

Ces fonctions nous permettrons d'illustrer la notion de flot monadique dans la monade IO sur les entrées et sorties standards.

3 Flots d'actions monadiques

Nous avons maintenant tous les ingrédients disponibles pour définir et manipuler, en OCaml, la notion de flots d'actions monadiques ou, plus simplement, flots monadiques.

3.1 Flots monadiques

Dans un contexte de programmation réactive, une façon simple de représenter les flots de données, finis ou infinis, consiste à définir un flot de type `'a s`, c'est-à-dire un flot contenant des valeurs de type `'a`, comme l'encapsulation d'une action monadique dont l'exécution renvoie :

(1) soit la valeur `None` lorsque le flot est *terminé*,

(2) soit `Some (a , sc)` lorsque le flot *produit* la valeur `a` et *continue* comme `sc`.

Cette définition, augmentée de quelques fonctions auxiliaires, est formalisée en OCaml par l'interface suivante :

```
module type STREAM  = functor (M : MONAD)  ->
  sig
    type 'a s = Stream of ('a * 'a s) option M.m
    val exec : 'a s -> unit M.m
    val fromStream : 'a s -> ('a * 'a s) option M.m
    val toStream : ('a s) M.m -> 'a s
    val fmap : ('a -> 'b) -> 'a s -> 'b s
  end
```

instanciée par :

```
module Stream : STREAM = functor (M : MONAD) ->
  struct
    type 'a s = Stream of (('a * ('a s)) option) M.m
    let rec exec (Stream m) = M.bind m (function
          | None -> M.return ()
          | Some (a,sc) -> exec sc)
    let fromStream (Stream m) = m
    let toStream ms = Stream (M.bind ms fromStream)
    let rec fmap f (Stream m) = Stream (M.bind m  (function
         | None -> M.return None
         | Some (a,sc) -> M.return (Some ((f a),(fmap f sc)))))
  end
```

Le constructeur de type `s` du module `Stream(M)` défini ci-dessus est connu pour être le *transformateur de monade* associé aux listes inductives. Appliqué au foncteur identité, il ne fait que reproduire le type liste.

La fonction `fromStream` permet simplement d'accéder, en évitant un « pattern matching », à l'action définissant le flot monadique. La fonction `toStream : 'a s m -> 'a s`, qui n'est pas réciproque de la précédente, permet de transformer une action produisant un flot en le flot qu'elle produit. Ces deux fonctions seront surtout utilisées pour simplifier notre code.

La fonction `fmap` est un exemple archétype de constructeur de fonctions synchrones (voir section suivante). Elle illustre en particulier la façon dont les flots monadiques codent, tout à la fois, des flots d'entrées comme des flots de sorties. En effet, un flot monadique peut être vu :

(1) comme un flot d'entrée lorsqu'on « exécute » récursivement les actions qui le compose,

(2) comme un flot de sortie lorsqu'on « produit » récursivement les actions qui le compose.

D'une certaine façon, dans le code de `fmap` c'est la fonction `bind` qui réalise cette correspondance entre entrées et sorties, le premier argument du bind étant une action d'entrée, à exécuter, dont le résultat est transmis à son deuxième argument qui produit l'action de sortie. La fonction

`fmap` montre aussi que le constructeur de type `s` peut être vu comme un foncteur de type. On peut en effet vérifier que les équations (4) et (5) sont satisfaites.

3.2 Fonctions synchrones dérivées

Nous dirons qu'une fonction sur les flots de données est synchrone lorsque tous ses flots d'entrée et son flot de sortie sont lus/produits à la même cadence. Comme déjà illustré ci-dessus par la fonction `fmap`, définir une fonction synchrone, revient à associer chaque action(s) de(s) flot(s) d'entrée à une action du flot de sortie. Des exemples de fonctions synchrones sont définis dans le module suivant :

```
module SStream (M : MONAD) =
  struct
    include Stream(M)
```

La fonction `zip : 'a s -> 'b s -> ('a * 'b) s` permet de fusionner, de façon synchrone, deux flots d'entrées, en utilisant l'opérateur `>>=`, version infixe de la fonction `bind` :

```
    let rec zip (Stream m1) (Stream m2) =
      Stream (m1 >>= (function
         | None -> return None
         | Some (a1,sc1) ->
            m2 >>= (function
               | None -> return None
               | Some (a2,sc2) ->
                  return (Some ((a1,a2),zip sc1 sc2)))))
```

Remarquons que, si les actions de ces flots d'entrées, supposées bloquantes, sont cadencées sur des horloges non synchronisées, le flot de sortie du zip est cadencé à chaque instant sur l'horloge la plus lente ; cette horloge de sortie peut être vue comme la borne supérieures des horloges d'entrées.

La fonction `iterate : 'a M.m -> 'a s` permet de créer un flux infini obtenu par itération d'une action d'entrée. La fonction `iterateSome : 'a option M.m -> 'a s` est une variante de la fonction précédente produisant un flot qui s'arrête dès que l'action itérée retourne `None`.

```
    let rec iterate m
      = Stream (m >>= (fun x -> return (Some (x, iterate m))))
    let rec iterateSome m
      = Stream (m >>= (function
              | None -> return None
              | Some x -> return (Some (x, iterateSome m))))
```

La fonction `loop : 's -> ('a * 's -> 'b * 's)-> 'a s -> 'b s` permet de coder n'importe quelle machine de Mealy.

```
    let rec loop s f (Stream m)
      = Stream (m >>= (function
           | None -> return None
           | Some (a,sc) ->  (f (s,a)) >>= (function
               | (s',b) -> return (Some (b, loop s' f sc)))))
  end
```

En effet, dans un appel `loop s f`, le premier argument `s : 's` définit l'état initial de la ma-

chine, le second argument `f : 'a * 's -> 'b * 's` définit tout à la fois la fonction d'entrée/-sortie et la fonction de mise à jour de son état courant.

3.3 Application aux flots d'IO : module `streamIO`

Appliqué à la monade IO, on obtient le module `StreamIO` qui permet d'illustrer le concept de flot monadique sur les entrées et sorties standards.

```
module StreamIO = SStream (IO)

let stdinStream = StreamIO.iterate getChar
```

La fonction d'écho de l'entrée standard sur la sortie standard peut alors être codée par :

```
let printStream s = StreamIO.exec (StreamIO.fmap print_char s)

let echo = IO.run (printStream (stdinStream))
```

On peut aussi lui préférer une version qui termine sur l'entrée d'un caractère particulier `c` agissant comme marque de fin :

```
let stdinStreamUntil c
  = StreamIO.iterateSome
      (IO.bind getChar
         (fun x -> if (x == c) then IO.return None
                   else IO.return (Some x)))

let echoUntil c = IO.run (printStream (stdinStreamUntil c))
```

Dans les deux cas, l'expérimentation de ces fonctions d'échos montre qu'elles s'exécutent sans fuite de mémoire.

4 Des flots aux promesses de flots

Lire le flot d'entrées à travers le flot monadique `stdinStream` signifie, sans surprise, exécuter de façon itérative l'action `getChar`. Cela implique que deux processus partageant ce même flot `stdinStream` feront, chacun, des appel à `getChar`. Les valeurs de ce flot d'entrée ne seront donc pas partagées par ces deux processus, mais, on contraire, distribuées entre ces processus, chacun exécutant de façon indépendante l'action `getChar`. C'est avec une notion de promesse de flot, qui se codera par un flot de promesses, que nous remédierons à cela : les promesses de flots, comme les promesses usuelles, permettant de partager le résultat d'une action sans reproduire ses effets de bord.

Plus précisément, nous définissons dans un premier temps une notion de référence monadique dont nous dériverons la notion de promesse telle qu'elle apparait dans les bibliothèques `async` et `lwt`. Défini au sein d'une monade, nous pouvons aussi axiomatiser quelques unes des propriétés de ces références monadiques, nous permettant alors de prouver que les promesses qui en découlent forment bien une monade. Appliquées aux flots monadiques, ces références monadiques nous permettent de définir les promesses de flots annoncées et, in fine, des fonctions asynchrones sur ces flots.

4.1 Références et promesses monadiques

Une référence monadique est associée à une action monadique en cours d'exécution. Elle permet d'attendre puis d'accéder, quand elle est disponible, à la valeur retournée par cette action. En OCaml, nous pouvons la définir de façon générique avec l'interface :

```
module type MONADREF =
  sig
    include MONAD
    type 'a mref
    val fork : 'a m -> 'a mref m
    val read : 'a mref -> 'a m
  end
```

L'action `fork m` permet de lancer l'action `m` passée en paramètre et retourne une référence à cette action. L'action `read r` permet au contraire d'attendre et de lire la valeur produite par l'action référencée par `r`.

La sémantique élementaire de ces références est capturée par les trois équations suivantes qui devront être satisfaites par toutes instances.

$$\texttt{(fork m)>>= read} \equiv \texttt{m} \tag{6}$$

$$\texttt{fork} \circ \texttt{read} \equiv \texttt{return} \tag{7}$$

$$\texttt{fork(m >>= f)} \equiv \texttt{(fork m)>>= fun r -> fork(read r >>= f)} \tag{8}$$

pour tout action `m : 'a m`, et toute fonction `f : 'a -> 'b m`. C'est le résultat de la fonction fork, de type `'a mref m`, qui est défini comme une *promesse*, l'équation (6) ci-dessus explicitant la composition monadique (bind) d'une promesse a un comportement équivalent à celui de l'action qui a été lancée. Le lecteur intéressé pourra consulter l'étude détaillée [6] de la notion de référence monadique.

4.2 Exemple de références d'IO : le module `IO_Ref`

Les références d'IO sont construites à l'aide de `MVar` [8], définit grâce aux `Mutex` et `Control` de l'extension `threads` d'OCaml. La création de références d'IO (et donc de promesses d'IO) passe par le lancement de threads indépendants qui vont mettre à jour ces promesses.

```
module IO_Ref : MONADREF
  with type 'a m = 'a io and type 'a mref =  'a MVar.mvar
  =
  struct
    include IO
    type 'a mref = 'a MVar.mvar
    let rec fork m
      = let v = MVar.createEmpty () in
        ignore (Thread.create (fun _ -> MVar.put v (run m)) ());
        return v
    let read v = return (MVar.read v)
  end
```

Les `mvar` permettent la communication entre les threads. Une `mvar` ne peut être que dans deux états : vide ou pleine. La fonction `put` met une valeur dans une `mvar`, en bloquant tant qu'elle

est pleine. La fonction `read` permet de lire la valeur d'une `mvar` sans la modifier. La fonction `take` permet de lire la valeur d'une `mvar` en la laissant vide. Ces deux fonctions sont bloquantes tant que la `mvar` est vide.

Cette instance non triviale de notre définition de références monadiques permet de vérifier, sur toutes sortes d'exemples concrets, la validité des équations (6)–(8).

4.3 Les promesses forment-elles une monade ?

Dans les deux bibliothèques `async` et `lwt` d'OCaml qui implémentent les promesses, ces dernières sont présentées à travers le prisme de la programmation monadique puisque c'est une fonction `bind` qui permet d'associer une *call-back* à une promesse. Est-ce une convention d'écriture ou l'existence réelle d'un foncteur monadique sur les types induit par les promesses ?

Nous avons défini ci-dessus les promesses comme les actions renvoyées par la fonction `fork`. On retrouve une interface analogue à ces deux bibliothèques en posant :

```
module type ASYNC = functor (M : MONADREF) ->
  sig
    type 'a p = 'a M.mref M.m
    val return : 'a -> 'a p
    val bind : 'a p -> ('a -> 'b p) -> 'b p
    val fmap : ('a -> 'b) -> 'a p  -> 'b p
  end
```

instancié par :

```
module Async : ASYNC = functor (M : MONADREF) ->
  struct
    type 'a p = 'a M.mref M.m
    let return a = M.fork (M.return a)
    let bind m f = M.bind m
          (fun r -> M.fork (M.bind (M.bind (M.read r) f) M.read))
    let fmap f m = bind m (fun r -> return (f r))
  end
```

Grâce aux équations (6)–(8) on peut vérifier que la fonction `fmap` satisfait bien les équations (4)–(5) prouvant ainsi que la fonction de type `'a -> 'a p` induite par une monade avec références `M` est bien un foncteur de type, dans le *cas général* d'actions monadiques de type `'a mref m`. On peut aussi formellement prouver que la fonction `fmap` peut être directement définie dans la monade `M` par :

$$\texttt{Async.fmap m f} \equiv \texttt{fork (m >>= fun r -> bind (read r)>>= (return} \circ \texttt{f))} \quad (9)$$

dès lors que les propriétés (6)–(8) sont satisfaites. Par ailleurs, tout comme dans la bibliothèque `async` la fonction `Async.bind` définie ici permet d'associer à une promesse `m` une fonction de *call-back* qui sera déclenchée à la réalisation de la promesse. Les fonctions `return` et `bind` définies ci-dessus définissent-elle pour autant une monade de promesses ? La réponse à cette question est positive *à condition* de se limiter à la notion de promesses définies ci-dessus.

Plus précisément, dans le cas général d'actions de type `'a p`, on peut vérifier que les équations (2)–(3) sont satisfaites. Par contre, l'équation (1) ne l'est pas. Il faut se restreindre aux fonctions `f : 'a -> 'b p` qui renvoient bien des promesses, c'est-à-dire des fonctions `f` de la forme `fork` $\circ$ `f1` avec `f1 : 'a -> 'b m`.

En effet, un raisonnement simple montre que la validité de l'équation (1) dans le cas général implique, dans la monade `m` étendue par des références, l'équivalence

```
fork (bind m read) ≡ m
```

pour toute action `m : 'b mref m`. Mais cette équivalence admet un contre exemple dans n'importe quelle extension de type `MONADREF` de la monade IO en posant :

```
m = bind readChar (fun c -> fork (return c))
```

on constante en effet que l'action `m` bloque tant qu'un caractère n'est pas entré sur l'entrée standard, alors que l'action `fork(bind m read)` renvoie sans attendre une référence monadique. N'ayant pas les mêmes effets de bord, ces deux actions ne peuvent être équivalentes, bien qu'*in fine* elles renvoient la promesse du même caractère d'entrée.

Plus encore, on peut vérifier que la notion de référence d'action monadique induit une sorte de catégorie de Kleisli dont les objets sont les types et les flèches de la forme `F = fork ∘ f1` pour toute fonction `f1 : ' a -> 'b m` de la catégorie de Kleisli associée à la monade `m`, avec la composition définie par :

```
g ∘ f = fun a -> fork((f a) >>= read >>= g >>= read)
```

Modulo les équivalences d'actions monadiques induites par (1)–(3) pour la monad `m` et les équivalences de promesses induites par (6)–(8) pour son extension par des références, il apparait que ces deux catégories de Kleisli sont *isomorphes*.

Autrement dit, on peut prouver formellement que la programmation par promesses définies dans une monade `m` est *structurellement* équivalente à la programmation par actions monadiques dans cette même monade.

4.4 Flots de promesses

Nous souhaitons maintenant définir une extension de cette notion de promesses applicables aux flots monadiques. Plus précisément, nous souhaitons disposer d'un mécanisme permettant de lancer l'évaluation d'un flot et obtenir en retour un accesseur aux valeurs portées par ce flot, c'est-à-dire une promesse de flot.

Pour ce faire, nous transformons la définition de nos flots monadiques en une définition de flots de références monadiques.

```
module StreamRef (M : MONADREF)
  =
  struct
    include SStream(M)
    open M
    type 'a sref
       = StreamRef of (('a * ('a sref)) option) mref
```

On peut observer l'analogie entre les inductions définissant les flots dans `Stream` et `StreamRef`. Elles ne diffèrent en effet que par le nom de leur constructeur et le constructeur de type utilisé en paramètre : `m` pour `Stream` et `mref` pour `StreamRef`. La définition du module `StreamRef(M)` se poursuit avec :

```
    let forkStream s =
      let rec evalAndFork (Stream m)
        = m >>= function
              | None -> return None
              | Some (a,sc) -> (fork (evalAndFork sc)) >>=
                   (fun r -> return (Some (a, StreamRef r)))
      in (fork (evalAndFork s)) >>=
            (fun r -> return (StreamRef r))
```

de type `'a s -> 'a sref m` qui permet de créer une promesse de flot en lançant en quelque sorte l'évaluation du flot argument, et

```
    let rec readStream (StreamRef r)
      = Stream ((read r) >>= function
               | None -> M.return None
               | Some (a,rc) ->
                  M.return (Some (a, readStream rc)))
  end
```

de type `'a sref -> 'a s` qui permet de (re)lire, autant de fois que nécessaire, le flot référencé. On peut vérifier qu'on a bien :

$$\texttt{(forkStream s) >>= (return} \circ \texttt{readStream)} \quad \equiv \quad \texttt{return s} \tag{10}$$

pour tout flot monadique `s : 'a s`. On peut donc formellement définir une promesse de flot de type `'a s` comme le résultat de la fonction `forkStream` de type `'a sref m`, le bind de la monade sous-jacente nous permettant d'associer une call-back non pas à un seul évènement promis, mais aux flots d'évènements promis par cette promesse de flot.

Remarquons que, tout comme avec les références monadiques, alors que l'évaluation d'un flot peut avoir des effets de bord, sa relecture à travers une promesse n'en a essentiellement aucun, sauf à attendre les valeurs promises par le flot lancé.

Ces promesses de flots se révèlent aussi très utiles pour dupliquer le contenu d'un flot sans dupliquer les effets de bord associés à l'exécution des actions monadiques qui le définissent. On peut en effet tester la correction de toutes les fonctions ci-dessus en posant :

```
let echoRZUntil c = IO.run (printStream(toStream
     (forkStream (stdinStreamUntil c) >>=
        fun r -> return (readStream r))))

let print_double_char = fun (c1,c2) ->
  print_char '(' ;print_char c1 ;print_char ',';
  print_char c2 ;print_char ')'

let printDoubleStream s
  = exec (streamIO.fmap  print_double_char s)

let echo2RZUntil c = IO.run (printDoubleStream(toStream
     (forkStream (stdinStreamUntil c) >>=
        fun r -> return (zip (readStream r) (readStream r)))))
```

qui effectue, sans perte de mémoire observable, un double echo de l'entrée standard.

5 Un peu plus de concurrence

L'ajout de promesses et de concurrence ne saurait être complet sans offrir, de plus, la possibilité de faire des lectures de promesses non bloquantes et de trier les promesses par ordre de terminaison.

Comme nous allons le voir, cela va nous permettre d'enrichir notre bibliothèque de flots monadiques par des fonctions asynchrones nous permettant de faire, comme annoncé en introduction, de la programmation GALS.

5.1 Promesses concurrentes

L'interface de ces promesses un peu plus concurrentes est définie par :

```
module type  MONADREF_PLUS =
  sig
    include MONADREF
    val tryRead : 'a mref -> 'a option m
    val parRead : 'a mref -> 'b mref -> ('a,'b) sum m
  end
```

avec le constructeur de type somme défini par :

```
type ('a ,'b) sum =
  | Left of 'a
  | Right of 'b
let toLeft a = Left a
let toRight b = Right b
```

où la fonction `parRead` doit renvoyer, soit la valeur retournée par l'action terminant la première, soit n'importe laquelle des deux valeurs retournées si les deux actions sont déjà terminées, ou terminent en même temps. Grâce à la notion de `MVar`, nous pouvons instancier cette interface comme une extension de notre monade IO en posant :

```
module IO_Ref_Plus : MONADREF_PLUS =
  struct
    include IO_Ref
    let tryRead v =
      IO.return (MVar.tryRead v)
    let rec parRead  pl pr  =
      let v = MVar.createEmpty () in
      let tryWrite (p,f) =
        MVar.tryPut v (f (IO.run (read p)))
      in (ignore (Thread.create tryWrite (pl,toLeft));
          ignore (Thread.create tryWrite (pr,toRight));
          read v)
  end
```

5.2 Fusion asynchrone de flots monadique

La possibilité offerte par la fonction `parRead` de trier des actions référencées par ordre de terminaison nous conduit à notre première définition de fonctions asynchrones : la fusion temporelle

de deux flots monadiques. Plus précisément, on peut définir de façon générique :

```
module StreamRef_Plus  (M : MONADREF_PLUS)
  =
  struct
    include StreamRef(M)

    let fmapStream = fmap
    open M
    let rec mergeStreamRef (StreamRef r1) (StreamRef r2)
      = Stream ((parRead r1 r2) >>= function
          | Left None -> fromStream (readStream (StreamRef r2))
          | Left (Some(a1,sc)) -> return (Some (a1,
             mergeStreamRef sc (StreamRef r2)))
          | Right None -> fromStream (readStream (StreamRef r1))
          | Right (Some(a2,sc)) -> return (Some (a2,
             mergeStreamRef (StreamRef r1) sc)))
    let rec merge s1 s2
      = toStream (forkStream s1 >>= fun r1 ->
                  forkStream s2 >>= fun r2 ->
                  return (mergeStreamRef r1 r2))
    let mergeStream s1 s2
      = merge (fmapStream toLeft s1) (fmapStream toRight s2)
  end
```

La fonction `merge : 'a s -> 'a s -> 'a s` prend deux flots monadiques en argument et les fusionne, par ordre d'arrivée des valeurs qu'ils portent, pour produire le flux monadique résultant de cette fusion.

Cette fonction `merge` est l'archétype de ce qu'on pourrait appeler une fonction asynchrone, puisque l'horloge du flot de sortie est égale à l'union des horloges des flots d'entrées, certains ticks pouvant être dupliqués en cas de réceptions synchrones. Elle est à mettre en regard de la fonction `zip` qui, tout au contraire, produit un flot dont l'horloge est égale aux horloges des flux d'entrées, quitte à « forcer » l'égalité des horloges des flux d'entrées en prenant leur borne supérieure.

On peut vérifier que la fusion induit une structure de monoïde commutatif sur les flots monadiques, le flot vide défini par `emptyStream = Stream (return None)` servant de neutre [3]. A partir de ce merge, on peut aussi définir une monade avec le constructeur de type flot monadique, la fonction `bind s f` remplaçant chaque élément `a`, porté et placé dans le temps par le flot `s`, par son image `f a`, ces flots images étant alors fusionnés à la volée.

La fonction `merge` permet aussi de coder `readAll : 'a mref list -> 'a stream` qui, appliquée à une liste de références monadiques, produit le flot des valeurs retournées par les actions référencées, triées temporellement. Une telle fonction est très utile pour faire, par exemple, un « foldmap » concurrent. Néanmoins, ce codage, uniforme, est de complexité quadratique en nombre d'appel à `parRead`. Dans la monade IO on pourra donc lui préférer une version lineaire obtenu simplement en généralisant le code de `parRead`. Plus généralement, la fonction `parRead` dérivant facilement de toute implémentation de la fonction `readAll`, on pourra lui préférer cette

3. Techniquement, le flot vide défini en OCaml n'est que faiblement polymorphe. On peut remédier à cela en spécifiant le constructeur de type monadique `m` comme covariant, mais cela nous interdit alors de définir des monades via des structures mutables tels que les `MVar`.

dernière dans la signature `StreamRef_Plus`.

Le cœur de notre expérimentation de programmation GALS, combinant le contrôle asynchone de traitement audio synchrone [7] peut alors être définit de la façon suivante :

```
module AStream (M : MONADREF_PLUS) =
  struct
    include StreamRef_Plus(M)
    open M
    let rec asyncMapRef f (StreamRef mf) s =
      let applyAndCont f sf (Stream m) = m >>= (function
        | Some(a,sc) -> return (Some (f a , asyncMapRef f sf sc))
        | None -> return None) in
      Stream (tryRead mf >>= (function
        | None -> applyAndCont f  (StreamRef mf) s
        | Some (None) -> fromStream (fmapStream f s)
        | Some (Some(f1 , scf)) -> applyAndCont f1 scf s))
    let  asyncMap f sf s = toStream
       (forkStream sf >>= fun r -> return(asyncMapRef f r s))
  end
```

La fonction `asyncMap : ('a -> 'b)-> ('a -> 'b)s -> 'a s -> 'b s` reçoit en effet, de façon asynchrone, un flot de fonctions de type `'a -> 'b`, et applique de façon synchrone la dernière fonction arrivée (où la première donnée en paramètre) à son dernier paramètre, une flot de type `'a s`.

6 Performances comparées de promesses de flots

La fonction d'écho et ses variantes nous permettent de tester les performances de notre implémentation décrite ici et de la comparer à celle déjà réalisée en Haskell. En Haskell nous utilisons, aux choix, les threads léger de sa bibliothèque concurrente ou les thread légers du système unix. Dans la version OCaml proposée ici, nous n'utilisons que les threads légers Unix. Une autre version, dont nous rapportons aussi les tests, utilise les threads légers de la bibliothèque `lwt`.

	type thread	echo	echoR	echo2RZ
Haskell	Light	375 kHz	21 kHz	16.5 kHz
Haskell	System	375 kHz	4.9 kHz	4.8 kHz
OCaml	System	653 kHz	7.34 kHz	7.3 kHz
OCaml	Lwt	653 kHz	90 kHz	72 ,7 kHz

Nos tests de performance, effectué sur un ordinateur portable standard, consistent à faire un « pipe » unix d'un programme énumérant les entiers et d'observer, après 20 seconde, le nombre d'entiers traités par les programmes d'échos testés, rapporté à une seconde. Trois programmes d'échos sont testés, qui effectuent, selon le cas, un echo simple (`echo`) sans threads, un echo avec une promesse du flot d'entrée et une seul lecture (`echoR`) et, enfin, un echo avec une promesse du flot d'entrée et deux lectures zippées (`echo2RZ`). Les codes sont compilés avec `ghc -O3` pour Haskell et `ocamlopt -O3 I +threads` pour OCaml.

On constate, peut-être sans surprise à cause de l'évaluation paresseuse en Haskell, que les versions OCaml sont sensiblement plus rapides que les versions Haskell, les version avec threads légers étant aussi plus performantes que les versions avec threads systèmes. Remarquons cependant que la bibliothèque `lwt` offre bien moins de service que la librairie concurrente d'Haskell.

7 Conclusion

La programmation concurrente asynchrone a pour avantage, par rapport à la programmation concurrente généralisée, d'interdir par construction tout programme avec interblocages. Son applicabilité s'en trouve limitée ainsi limité. Néanmoins, combinée avec la programmation par flots monadiques et la notion de promesses de flots qui en découle, ces limites restent confortable pour de nombreuses applications. Notre étude permet ainsi d'identifier une première série de *primitives* asynchrones qui offrent, tout à la fois, la souplesse de la concurrence et l'assurance de programmes sans interblocages.

Cette étude n'est cependant pas terminée. En effet, il serait intéressant de pouvoir typer les actions monadiques comme étant bloquantes (actions d'entrée) ou non bloquantes (actions de traitement ou de sortie) afin de développer un système de type nous permettant de vérifier la cohérence temporelle des fonctions sur les flots monadiques. Des types modaux [9] pourraient être utilisés pour cela. La programmation OCaml est aussi, avec ses effets de bord, une programmation implicitement de type monadique. Il pourrait être interessant d'assumer ce fait en intégrant toutes les interfaces proposées ici au top-level de la programmation OCaml, en évitant ainsi le recours à des appels explicites de `return` et `bind`.

Références

[1] S. Archipoff and D. Janin. Structured reactive programming with polymorphic temporal tiles. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM Press, 2016.

[2] S. Archipoff and D. Janin. Unified media programming : An algebraic approach. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM, 2017.

[3] C. Deleuze. Concurrence légère en ocaml : muthreads. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2013.

[4] Robert H. Halstead, Jr. Multilisp : A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4) :501–538, 1985.

[5] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. A history of Haskell : Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*. ACM Press, 2007.

[6] D. Janin. An equational modeling of asynchronous concurrent programming. Technical report, LaBRI, Université de Bordeaux, 2019.

[7] D. Janin. Screaming in the IO monad. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM, 2019.

[8] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Principles of Programing Languages (POPL)*, New York, 1996. ACM.

[9] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Int. Conf. Func. Prog. (ICFP)*, 2013.

[10] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system, release 4.08, Documentation and user's manual*. Inria, 2019.

[11] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

[12] Y. Minsky, J. Hickey, and A. Madhavapeddy. *Real World Ocaml : Functional programming for the masses*. O'Reilly, 2013.

[13] E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science (CTCS)*, volume 530 of *LNCS*. Springer-Verlag, 1991.

[14] P. Wadler. Comprehending monads. In *Conference on LISP and Functional Programming (LFP)*, New York, 1990. ACM.

Programmation d'Applications Réactives Probabilistes

Guillaume Baudart[1], Louis Mandel[1], Marc Pouzet[2],
Eric Atkinson[3], Benjamin Sherman[3], Michael Carbin[3]

[1] MIT-IBM Watson AI Lab, IBM Research
[2] École normale supérieure, PSL University
[3] Massachusetts Institute of Technology

Abstract

Les langages synchrones ont été introduits pour concevoir des systèmes embarqués temps-réel. Ces langages dédiés permettent d'écrire une spécification précise du système, de la simuler, la valider par du test ou de la vérification formelle puis de la compiler vers du code exécutable. Cependant, ils offrent un support limité pour modéliser les comportements non-déterministes qui sont omniprésents dans les systèmes embarqués.

Dans cet article, nous présentons ProbZélus, une extension probabiliste d'un langage synchrone descendant de Lustre. ProbZélus permet de décrire des modèles probabilistes réactifs en interaction avec un environnement observable. Lors de l'exécution, un ensemble de techniques d'inférence peut être utilisé pour apprendre les distributions de paramètres du modèle à partir de données observées. Nous illustrons l'expressivité de ProbZélus avec des exemples comme un détecteur de trajectoire à partir d'observations bruitées, ou un contrôleur de robot capable d'inférer à la fois sa position et une carte de son environnement.

1 Introduction

Les langages synchrones ont été introduits il y a 30 ans pour la conception de systèmes embarqués temps-réel. Ils sont fondés sur le modèle de *parallélisme synchrone* [3]. De nombreux langages ont été proposés, dont Scade [8] utilisé pour concevoir et implémenter les logiciels temps-réel critiques (commandes de vol, freinage, contrôle moteur, par exemple).

Scade est un langage *flot de données* descendant de Lustre : les signaux d'entrées/sorties sont des suites infinies (ou *flots*), un système (ou *nœud*) est une fonction de suites, et toutes les suites progressent ensemble, instant après instant, de manière *synchrone*. Ce style de programmation est bien adapté pour exprimer les blocs de contrôle classiques (relais, filtres, contrôleurs PID, etc.), un modèle discret de l'environnement et une boucle d'interaction entre ces deux composants. Le code suivant implémente un contrôleur PID (proportionnel, intégral, dérivé) en Zélus, un langage académique proche de Scade [5].[1]

```
let node pid (r, y) = u where
  rec e = r -. y
  and u = p *. e +. i *. integr(0., e) +. d *. deriv(e)
```

Le nœud `pid` définit un flot de commandes `u` à partir d'un flot de consignes `r` et d'un flot de mesures `y`. La commande est la somme pondérée de trois actions (proportionnelle, intégrale, et dérivée) appliquée à l'erreur entre la consigne et la mesure. Les poids `p`, `i`, et `d` sont des constantes et les appels de nœuds `integr(0., e)` et `deriv(e)` calculent respectivement l'intégrale (initialisée à `0.`) et la dérivée de `e`.

Les langages synchrones offrent un support limité pour modéliser le non-déterminisme et les incertitudes de l'environment, pourtant omniprésents dans un système réel. Un contrôleur n'a souvent qu'une vision partielle et bruitée de son environnement, et le comportement du

[1] www.zelus.di.ens.fr

système lui-même est souvent sujet à des perturbations. Il est bien sûr possible de programmer des contrôleurs arbitrairement compliqués, capable d'anticiper et de compenser ces incertitudes, mais leur programmation est notoirement difficile et source d'erreurs [1].

Les langages de programmation probabilistes permettent de décrire des modèles probabilistes et d'*inférer* automatiquement les distributions de paramètres *latents* (i.e., non-observés) à partir d'*observations* (i.e., des entrées). Une approche populaire [4, 10, 16, 20–22] consiste à étendre un langage de programmation généraliste avec trois nouvelles constructions: (1) `x = sample(d)` introduit une variable aléatoire *latente* `x` de distribution `d`; (2) `observe(d, y)` mesure la *vraisemblance* d'une *observation* `y` par rapport à une distribution `d` (i.e., la densité de `y` par rapport à `d`); (3) `infer m obs` calcule la distribution des valeurs de sortie d'un programme ou *modèle* `m` *sachant* les observations données en entrée `obs`. Les langages de programmation probabilistes offrent une variété de techniques d'inférence automatique qui vont du calcul symbolique exact, aux approximations par échantillonnage (méthodes de Monte-Carlo). Mais aucun de ces langages n'offre le support et les garanties associées données par les langages synchrones pour concevoir des systèmes réactifs embarqués (programmation flot-de-données, exécution avec des ressources bornées, absence d'interbloquage).

Dans cet article, nous présentons ProbZélus [2], une extension probabiliste de Zélus. ProbZélus permet de combiner les constructions d'un langage réactif synchrone et les constructions d'un langage probabiliste — `sample`, `observe` et `infer` — pour développer des *applications réactives probabilistes*. Nous illustrons par des exemples les avantages offerts par ProbZélus :

1. Programmation de modèles probabilistes réactifs : un détecteur de trajectoire à partir d'observations bruitées (section 2).
2. Inférence dans la boucle : un contrôleur de robot guidé par le résultat d'un détecteur de trajectoire probabiliste (section 3).
3. Inférence semi-symbolique : un modèle de robot plus complexe, capable d'inférer à la fois sa position et une carte de son environnement (section 4).

2 Programmation Probabiliste Réactive

Nous rappelons ici le principe de l'inférence bayésienne sur lequel se fonde les langages de programmation probabilistes. Nous présentons ensuite un premier modèle probabiliste réactif : un détecteur de trajectoire à partir d'observations bruitées.

L'inférence bayésienne permet de calculer la probabilité d'une hypothèse (distribution *a posteriori*) à partir de croyances antérieures (distribution *a priori*), et d'observations. L'inférence s'appuie sur le théorème de Bayes :

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{x})p(\mathbf{y}|\mathbf{x})}{p(\mathbf{y})} \propto p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$$

La distribution *a posteriori* $p(\mathbf{x}|\mathbf{y})$ des variables *latentes* $\mathbf{x}$ après avoir observé les données $\mathbf{y}$ dépend essentiellement de deux termes : $p(\mathbf{x})$ la distribution supposée de $\mathbf{x}$ *a priori* (construction `sample`), et $p(\mathbf{y}|\mathbf{x})$ la *vraisemblance* des données $\mathbf{y}$ sachant $\mathbf{x}$ (construction `observe`).

En ProbZélus, les modèles probabilistes sont des nœuds particuliers introduits par le mot-clef `proba`. Les constructions `sample` et `observe` ne peuvent être invoquées qu'à l'intérieur d'un nœud probabiliste. L'opérateur `infer` prend en argument un nœud probabiliste et produit un resultat déterministe : la distribution *a posteriori* définie par le modèle. Ces contraintes sont vérifiées par typage lors de la compilation. Les signatures associées aux constructions probabilistes sont les suivantes :

```
val sample  : 'a Distribution.t ~D~> 'a
val observe : 'a Distribution.t * 'a ~D~> unit
val infer   : ('a ~D~> 'b) -S-> 'a -D-> 'b Distribution.t
```

Les flèches indiquent la nature des fonctions : `~D~>` indique un nœud probabiliste, `-D->` un nœud déterministe, et `-S->` indique un argument statique (une constante connue à la compilation). Le premier argument de `infer` est statique car ProbZélus limite l'ordre superieur aux fonctions de flots et n'accepte pas les flots de fonctions de flots. Le deuxième argument de `infer`, ainsi que les arguments de `sample` et `observe`, sont des flots de valeurs.

Méthodes d'inférence. Un programme probabiliste peut être compilé en un *échantillonneur* qui génère un échantillon aléatoire et un score qui mesure la qualité de cet échantillon (i.e., une exécution possible du programme et sa vraisemblance). Dans ce cadre, chaque `sample` tire aléatoirement une valeur dans la distribution associée, et chaque `observe` met à jour le score de l'échantillon. La méthode d'inférence la plus simple, *l'échantillonnage préférentiel*, lance N particules indépendantes. Chaque particule exécute l'échantillonneur pour obtenir une paire (valeur, score). Les résultats sont ensuite normalisés pour approximer la distribution *a posteriori*.

Malheureusement au cours de l'inférence, certaines particules empruntent des chemins d'exécution très improbables qui n'ont que peu d'influence et pénalisent donc l'estimation de la distribution *a posteriori*. Pour remédier à ce problème, les méthodes de *Monte-Carlo séquentielles* (MCS) (aussi appelé *filtres particulaires*) ré-échantillonnent périodiquement l'ensemble de particules au cours de l'exécution [9]. Les particules les plus improbables sont alors éliminées, et les plus probables sont dupliquées.

Enfin, la méthode d'*échantillonnage retardé* [15] permet de combiner des calculs symboliques exacts partiels avec des méthodes d'échantillonnage. En plus du score, les particules maintiennent un *réseau bayésien* qui capture symboliquement les distributions conditionnelles associées à un sous ensemble de variables aléatoires. Les observations peuvent être incorporées en conditionnant analytiquement le réseau. Les particules ne tirent un échantillon que si les calculs analytiques échouent, ou si une valeur concrète est nécessaire (e.g., condition d'un `if`).

Exemple. Considérons un robot qui cherche à estimer sa position courante à partir d'observations bruitées. La Figure 1 présente une modélisation possible de ce problème sous la forme d'un modèle de Markov caché (MMC). À chaque instant, la position courante x_t est une variable latente (cercle blanc) qui ne peut être observée directement. Le robot reçoit des observations y_t (cercle gris) produit par un capteur bruité, par exemple un radar. Chaque flèche indique une dépendance entre deux variables aléatoires. L'observation courante y_t dépend de la position courante x_t, et la position courante dépend de la position à l'instant précédent x_{t-1}. Le code ProbZélus correspondant est le suivant :

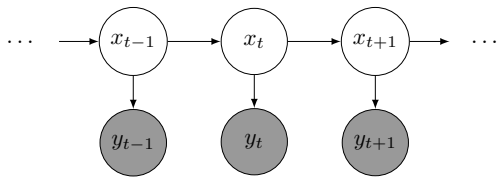


Figure 1: Un modèle de Markov caché. Les variables sont *latentes* (blanc) ou *observées* (gris).

```
let proba hmm (x0, y) = x where
  rec x =  sample (gaussian (x0 -> pre x, speed_x))
  and () = observe (gaussian (x, noise_x), y)

let node main x0 = display(y, x_dist) where
  rec y = sensor()
  and x_dist = infer hmm (x0, y)
```

Le nœud `main` illustre l'utilisation d'`infer` dans un nœud déterministe. Le corps de `main` définit deux flots: `y` les données bruitées envoyées par le `sensor`, et `x_dist` la suite de distributions de positions inférée à partir du modèle `hmm`, de la position initiale `x0` et des observations `y`. À chaque instant, la fonction `display` réalise l'affichage des flots `y` et `x_dist`.

Le nœud probabiliste `hmm`, introduit par le mot-clef `proba`, décrit le modèle de la Figure 1. La première équation indique que la position courante `x` est distribuée normalement autour de la position précédente (l'operateur d'initialisation `->` renvoie la valeur initiale `x0` au premier instant, et la position précédente `pre x` aux instants suivants). La seconde équation indique que l'observation courante `y` est distribuée normalement autour de la position courante `x`. Dans les deux cas, les variances des gaussiennes `speed_x` et `noise_x` sont des constantes.

3 Inférence dans la boucle

ProbZélus permet de mélanger arbitrairement du code Zélus déterministe avec du code probabiliste (à condition de respecter les contraintes de typage indiquées plus haut). L'inférence s'exécute en parallèle avec les processus déterministes. À chaque instant, les composants déterministes peuvent donc utiliser les résultats calculés par l'inférence. C'est ce que nous appelons *l'inférence dans la boucle*.

Pour illustrer cette approche, nous programmons un robot qui peut estimer sa position à partir des commandes qui lui sont données et de la lecture d'un GPS. Les commandes sont calculées en fonction de l'estimation de la position, c'est-à-dire du résultat de l'inférence.

Modularité. Les commandes reçues par le robot sont des accélérations. Pour estimer la position du robot à partir de ces accélérations, nous définissons un nœud `tracker` qui calcule un flot de position `p` et de vitesse `v` en intégrant un flot d'accélération `a` à partir des conditions initiales `p0` et `v0`.

```
let node tracker(p0, v0, a) = p where
  rec p = integr(p0, v)
  and v = integr(v0, a)
```

À cause de facteurs comme les frottements du moteur, l'adhérence des roues ou l'inclinaison du terrain, l'effet de la commande sur la position du robot n'est pas déterministe. Nous pouvons donc considérer ces commandes comme bruitées et utiliser le nœud probabiliste `hmm` présenté section 2 pour prendre en compte ce bruit. On peut ainsi estimer la position `p`, la vitesse `v` et l'accélération `a` à partir de la commande `u` en combinant le nœud probabiliste `hmm` avec le nœud déterministe `tracker`.

```
let proba acc_tracker(p0, v0, a0, u) = p where
  rec a = hmm(a0, u)
  and p = tracker(p0, v0, a)
```

Activation sporadique. Intégrer l'accélération pour estimer la position accumule les erreurs d'estimation. Ainsi, plus le temps passe, plus la position réelle du robot s'éloigne de la position estimée. Pour remédier à ce problème, le robot utilise également un GPS. Comme les mesures GPS sont coûteuses à réaliser, le robot appel le GPS de façon sporadique et s'appuie sur l'accélération pour estimer sa position entre deux mesures. Le nœud ProbZélus suivant intègre les mesures du GPS (également bruitées) au traqueur précédent.

```
let proba gps_acc_tracker(p0, v0, a0, u, gps) = p where
  rec p = acc_tracker(p0, v0, a0, u)
  and () = present gps(p_obs) -> observe(gaussian(p, p_noise), p_obs) else ()
```

À chaque instant, le nœud `acc_tracker` renvoie une estimation de la position courante `p`. L'entrée `gps` est un *signal* qui est émis quand le GPS mesure une nouvelle position. Quand la valeur `p_obs` est émise sur le signal `gps`, la construction `present` exécute son corps et conditionne alors le modèle avec l'observation de cette nouvelle donnée. La variance `p_noise` est une constante globale qui représente la variance des mesures du GPS.

Boucle de rétroaction. Maintenant que nous avons un modèle qui permet d'estimer la position courante du robot, nous pouvons utiliser la distribution de positions inférées pour mettre à jour la commande.

```
let node robot(p0, v0, a0, target) = (u, p_dist) where
  rec gps = geolocalizer ()
  and u = zero -> controller(pre (mean p_dist), target)
  and p_dist = infer gps_acc_tracker (p0, v0, a0, u, gps)
```

Le nœud `geolocalizer` génère le signal sporadique `gps`. Le nœud `controller` calcule la commande `u` à partir de la moyenne (`mean`) de la distribution de positions estimées `p_dist`. Ce flot de distribution `p_dist` est inféré à partir du nœud probabiliste `gps_acc_tracker` qui prend en entrée les conditions initiales, la commande `u`, et le signal `gps`. On observe donc bien une boucle de rétroaction entre le contrôleur et l'inférence. Les règles de causalité restent les même que pour Zélus: les cycles de dépendances doivent contenir un délai unitaire (`pre`). Cela n'empêche pas d'avoir dans un même instant du code probabiliste qui dépend de code déterministe et du code déterministe qui dépend de code probabiliste.

Structures de contrôle. ProbZélus offre de nombreuse structures de contrôle : signaux d'activation, ré-initialisation modulaire, et automates hiérarchiques [7]. Il est ainsi possible de programmer dans un formalisme proche des diagrammes par blocs [11], une notation classique des systèmes embarqués.

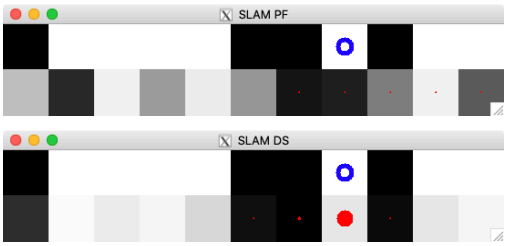
Nous avons vu avec le nœud `gps_acc_tracker` que les structures de contrôle comme `present` peuvent être utilisées à l'intérieur des nœuds probabilistes. Ces structures de contrôle peuvent également être utilisées à l'extérieur pour contrôler l'inférence. Par exemple, notre robot peut être utilisé pour effectuer une tâche lorsqu'il atteint une certaine position.

```
let node task_bot(p0, v0, a0, target) = cmd where
  rec automaton
  | Go -> do cmd, p_dist = robot(p0, v0, a0, target)
          until (probability p_dist (target - epsilon) (target + epsilon) > 0.9)
          then Task
  | Task -> do cmd = task_controller() done
```

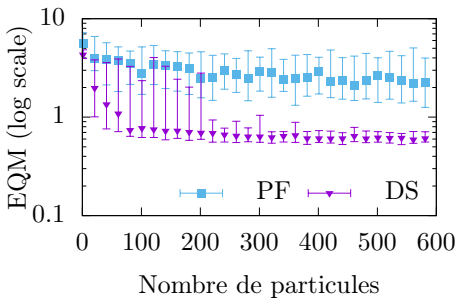
Dans l'état `Go`, la commande est celle calculée par le contrôleur `robot`, qui renvoie aussi la distribution de positions courantes. Lorsque la probabilité que le robot soit proche de la cible (entre `target - espilon`, and `target + epsilon`) est supérieure à 0.9, le contrôleur entre dans l'état `Task` où la commande est calculée par le nœud `task_controller`.

4 Inférence semi-symbolique

Les méthodes d'inférence par échantillonnage permettent d'obtenir des résultats satisfaisants pour les exemples précédents avec un nombre raisonnable de particules (≤ 1000). Malheureusement, ces méthodes peuvent échouer sur des modèles plus complexes où l'approche semi-symbolique de l'*échantillonnage retardé* peut donner de bons résultats.



(a) Pour chaque capture d'écran, la ligne supérieure montre la carte, et le cercle bleu la position exacte du robot. La ligne inférieure représente la carte inférée où le niveau de gris indique la probabilité pour la case d'être noire et les points rouges la probabilité de présence du robot sur la case.



(b) Précision en fonction du nombre de particules. Les points représentent la médiane de 40 exécutions, les barres d'erreur les 90% et 10% quantiles.

Figure 2: Exécution du SLAM avec filtre particulaire (PF) et échantillonnage retardé (DS).

Dans cette section, nous illustrons cette situation avec un contrôleur de robot capable d'inférer à la fois sa position courante et une carte de son environnement. Un problème classique de localisation et cartographie simultanées (*Simultaneous Location And Mapping*) [14].

SLAM. Considérons le cas simple où le robot évolue dans un monde discret à une dimension et chaque position correspond à une case noire ou blanche. Un robot peut se déplacer de gauche à droite et il peut observer la couleur de la case sur laquelle il se tient à l'aide d'un capteur. Il y a deux sources d'incertitude : (1) Les roues du robot sont glissantes, le robot peut donc parfois rester sur place en pensant se déplacer. (2) Le capteur fait des erreurs de lecture, et peut inverser les couleurs. Le contrôleur cherche à inférer la carte (couleur des cases) et la position courante du robot (Figure 2a).

Le robot maintient une carte où chaque case est une variable aléatoire qui représente la probabilité d'être noire ou blanche (niveau de gris dans la Figure 2a). La distribution *a priori* de ces variables aléatoires est uniforme entre 0 et 1 (une distribution Beta(1,1)) :

```
let proba beta_priors _ = sample (beta (1., 1.))
```

Le robot démarre de la position `x0` et reçoit à chaque instant une commande `Right` ou `Left`. Il se déplace alors vers la gauche ou la droite suivant la commande avec une probabilité de 10% de rester sur place (modélisée par une loi de Bernoulli de paramètre 0.1).

```
let proba move (x0, cmd) = x where
  rec slip = sample (bernoulli 0.1)
  and xp = x0 -> pre x
  and x = match cmd with
      | Right -> min max_pos (if slip then xp else xp + 1)
      | Left  -> max min_pos (if slip then xp else xp - 1)
      end
```

On modélise de la même manière le capteur avec un probabilité d'erreur de lecture de 10% :

```
let proba read obs = if sample (bernoulli 0.1) then not obs else obs
```

À chaque instant, le robot calcule sa position `x` et récupére la valeur de la carte associée à cette position `c`. On suppose alors que l'observation `o` suit une loi de Bernoulli paramétrée par `c`.

```
let proba slam (obs, cmd) = (map, x) where
  rec init map = Array.init (max_pos + 1) beta_priors ()
  and x = move (0, cmd)
  and o = read (obs)
  and c = Array.get map x
  and () = observe (bernoulli (c, o))
```

Évaluation. On peut constater sur la partie haute de la Figure 2a, que le SLAM est un modèle particulièrement difficile pour le filtre particulaire. Les résultats sont beaucoup plus convaincants sur la partie basse de la Figure 2a qui utilise l'échantillonnage retardé.

Plus précisément, la Figure 2b présente la précision de l'estimation de la position et de la carte après 1500 instants pour un robot qui se déplace de gauche à droite sur une carte de taille 11. La précision est définie comme la somme des erreurs quadratiques moyennes (EQM) de chacune des variables aléatoires du modèle.

Comparé au filtre particulaire, l'échantillonnage retardé est capable d'exploiter la relation de conjugaison entre la distribution *a priori* des cases de la carte (Beta), et les observations bruitées (Bernoulli) pour mettre à jour la distribution des cases en incorporant les observations de manière analytique. Ainsi, si $p(c) = \text{Beta}(\alpha, \beta)$ et $p(o|c) = \text{Bernoulli(c)}$, on obtient selon l'observation o : $p(c|o = \mathit{true}) = \text{Beta}(\alpha + 1, \beta)$, ou $p(c|o = \mathit{false}) = \text{Beta}(\alpha, \beta + 1)$.

En revanche, le calcul symbolique échoue pour la position du robot qui ne peut être estimée qu'à partir d'un ensemble de particules (il n'y a pas de relation de conjugaison entre la position courante et la position précédente). La courbe DS n'atteint donc sa précision maximale que pour $100-200$ particules. L'exemple du SLAM illustre donc l'avantage d'une méthode semi-symbolique combinant calculs exacts et échantillonnage.

5 Travaux connexes

Programmation probabiliste. Ces dernières années, les langages de programmation probabilistes ont suscité un intérêt croissant. Certains langages comme BUGS [13], Stan [6] ou Augur [12] offrent des techniques d'inférence optimisées pour un sous-ensemble contraint de modèles. D'autres comme WebPPL [10], Edward [21], Pyro [4] ou Birch [16] permettent de spécifier des modèles arbitrairement complexes. Par rapport à ces langages, ProbZélus peut être utilisé pour programmer des *modèles parallèles réactifs* et qui ne terminent pas, et où l'inférence s'effectue en interaction avec des composants déterministes.

Non-déterminisme dans les Langages réactifs. Lutin est un langage pour décrire et simuler des systèmes réactifs non-déterministes [19] mais ne permet pas d'inférer des paramètres à partir d'observations. ProPL [17] est un langage pour décrire des processus probabilistes qui évoluent dans le temps. Comparé à ProbZélus, ProPL se concentre sur une classe restreinte de modèles : les *réseaux dynamiques bayésiens* (DBN) et s'appuie sur les techniques d'inférence standard pour les DBNs. CTPPL [18] est un langage pour décrire des processus probabilistes en temps continu. La durée nécessaire à un sous-processus peut être spécifiée par un modèle probabiliste. Ces modèles ne peuvent pas être exprimés en ProbZélus qui repose sur le modèle de temps synchrone discret.

6 Conclusion

Modéliser des comportements non-déterministes est un aspect fondamental des systèmes embarqués qui évoluent dans des environnements bruités et incertains. Les langages synchrones pourtant introduits pour la conception de tels systèmes n'offraient jusqu'alors que peu de support pour prendre en compte l'incertitude.

Dans cet article nous avons illustré les avantages offerts par ProbZélus, le premier langage synchrone probabiliste. ProbZélus permet d'écrire des modèles probabilistes réactifs capables d'inférer des paramètres latents à partir d'observations. L'inférence s'exécute en interaction avec des composants déterministes ce qui permet de programmer des systèmes avec *inférence dans la boucle*. Enfin, ProbZélus offre plusieurs méthodes d'inférence automatique qui combinent calculs symboliques et échantillonnage.

Bibliographie

[1] Karl J Åström. *Introduction to stochastic control theory*. Courier Corporation, 2012.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. *CoRR*, abs/1908.07563, 2019.

[3] Gérard Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.

[4] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20:28:1–28:6, 2019.

[5] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control*, 2013.

[6] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–37, 2017.

[7] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software*, 2005.

[8] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *International Symposium on Theoretical Aspects of Software Engineering*, 2017.

[9] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, 2006.

[10] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2019-10-15.

[11] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[12] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[13] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The bugs project: Evolution, critique and future directions. *Statistics in medicine*, 28(25):3049–3067, 2009.

[14] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI National Conference on Artificial Intelligence*, 2002.

[15] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *AISTATS Proceedings of Machine Learning Research*, 2018.

[16] Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

[17] Avi Pfeffer. Functional specification of probabilistic process models. In *AAAI National Conference on Artificial Intelligence*, 2005.

[18] Avi Pfeffer. CTPPL: A continuous time probabilistic programming language. In *International Jont Conference on Artifical Intelligence*, 2009.

[19] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal of Embedded Sytems*, 2008.

[20] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language Anglican. In *Symposium on the Implementation and Application of Functional Programming Languages*, 2016.

[21] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

[22] Yi Wu, Lei Li, Stuart J. Russell, and Rastislav Bodík. Swift: Compiled inference for probabilistic programming languages. In *International Joint Conference on Artificial Intelligence*, 2016.

Vers une formalisation en Coq de la provenance de données

Véronique Benzaken[1], Sarah Cohen-Boulakia[1], Évelyne Contejean[2], Chantal Keller[1], and Rébecca Zucchini[1,3]

[1] LRI, Université de Paris Sud, CNRS (UMR8623) - Université Paris Saclay
[2] LRI, CNRS (UMR8623), Université de Paris Sud - Université Paris Saclay
[3] École Normale Supérieure Paris-Saclay - Université Paris Saclay
prénom.nom@u-psud.fr

Résumé

Dans de multiples domaines scientifiques, de nombreuses données sont générées quotidiennement et doivent être analysées. Dans ces processus d'analyse, les données initiales sont combinées à d'autres jeux de données massifs. Pour garantir une interprétation correcte des résultats de ces analyses de données, il est crucial de pouvoir retracer la provenance des données produites à partir des données initiales. La communauté des bases de données a proposé un cadre formel unifiant de «semi-anneaux de provenance». L'objectif de cet article est de certifier a posteriori la correction d'une provenance. Pour ce faire, nous proposons une formalisation en Coq fondée sur le modèle de semi-anneaux de provenance pour des analyses de données exprimées en algèbre relationnelle. Nous introduisons ici notamment une preuve d'adéquation de cette provenance avec l'interprétation usuelle de l'algèbre relationnelle. Il s'agit d'une première étape vers la formalisation de langages centrés données avec des garanties fortes de provenance.

1 Introduction

L'explosion du volume de données générées par les multiples capteurs et équipements utilisés parfois quotidiennement – objets connectés, séquenceurs d'ADN, satellites sondeurs – pose de nouveaux défis dans le processus d'analyse de données. En effet, le passage de ces données brutes à l'acquisition de nouvelles connaissances nécessite une analyse fine de ces données massives, qu'il faut notamment comparer, croiser et combiner à d'autres jeux de données. De nombreuses données intermédiaires et finales sont alors générées à leur tour lors du processus d'analyse. Pour comprendre et pouvoir interpréter correctement le résultat d'une analyse, l'expert a besoin de connaître sa *provenance*, c'est-à-dire qu'il a besoin d'accéder à des informations relatives aux jeux de données qui ont été impliqués lors de l'analyse. Sans provenance, l'interprétation des résultats finaux peut être faussée et la reproductibilité de l'analyse ne plus être assurée.

De nombreuses formalisations de la provenance ont été proposées dans la communauté «bases de données» notamment autour du concept de «why provenance» [7] déterminant l'ensemble des combinaisons de lignes nécessaires pour qu'une ligne résultat existe. Déterminer la provenance d'un résultat de requête revient alors à propager les annotations associées aux données utilisées et combinées pour construire ce résultat. Selon les modèles, les relations sont annotées à des niveaux variés (depuis le niveau de la relation jusqu'à celui du n-uplet) et les annotations peuvent revêtir des formes diverses [11, 21, 14]. Green *et al.* ont introduit en 2007 un cadre unifiant pour ces modèles de provenance proposant de représenter la provenance sous la forme de semi-anneaux (*Provenance semirings*) [15, 17]. Néanmoins ce modèle n'a pas aujourd'hui de forme opérationnelle sûre : étant donnée une analyse de données effectuée, il n'existe pas d'implémentation capable de générer automatiquement la provenance d'un résultat et de

certifier que cette provenance est correcte, point crucial pour assurer la bonne interprétation des résultats et la reproductibilité des analyses.

L'objectif de cet article est de garantir formellement la provenance des données impliquées dans une analyse. Bien que les résultats soient indépendants de l'assistant de preuves choisi, nous présentons la formalisation telle que nous avons effectuée en Coq. Plus précisément, nous nous intéresserons ici aux analyses effectuant des transformations de données, fondées sur l'algèbre relationnelle (sélection de données, combinaison - ou jointure - entre les données de plusieurs tables, ...). Ces analyses capturent un large panel d'analyses de données qui sont opérationnelles lorsqu'elles sont exprimées en SQL. Dans ce contexte, la provenance se représente comme une combinaison des provenances (des données) initiales : les opérateurs de l'algèbre combinent ces annotations grâce aux opérations du semi-anneau, dans un procédé similaire à l'interprétation abstraite. La formalisation présentée ici est exécutable et donc se base sur le modèle fini usuel de SQL.

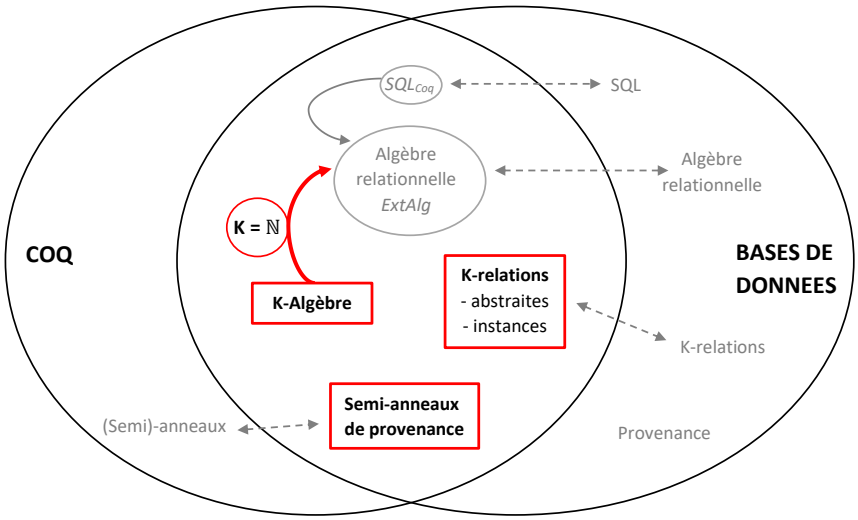


FIGURE 1 – Contributions

La Figure 1 introduit nos contributions et les place dans le paysage des bases de données et celui des bibliothèques de l'assistant Coq. Dans le domaine des bases de données, on retrouve la présence de l'algèbre relationnelle qui forme le noyau du langage SQL, les K-relations qui seront décrites plus précisément dans la Section 2 et qui constituent la base des semi-anneaux de provenance (relations annotées par des valeurs du semi-anneau K). Au niveau de Coq, on retrouve des bibliothèques dédiées aux semi-anneaux (indépendamment des K-relations) ainsi que SQLCoq et ExtAlg, des bibliothèques certifiées en Coq respectivement de SQL et de l'algèbre relationnelle [4, 3, 5].

Les contributions de l'article présent sont indiquées encadrées en trait gras (et en rouge). La première d'entre elles, présentée en Section 3, est la formalisation en Coq des semi-anneaux de provenance et de trois de leurs instances (entiers naturels, booléens, structures tropicales). La Section 4 présente notre formalisation en Coq des K-relations. La Section 5 présente notre théorème fondamental montrant l'adéquation des K-relations avec l'algèbre relationnelle classique, c'est-à-dire que la ℕ-Algèbre coïncide avec ExtAlg. Nous comparerons ensuite nos travaux avec l'état de l'art (Section 6) avant de conclure (Section 7).

2 Provenance et bases de données

2.1 Contexte

Une proposition pour être capable de suivre le cheminement des données au cours de transformations de données est d'utiliser des bases de données annotées. Chaque annotation contient des informations qui peuvent être de diverses natures : par exemple pour compter le nombre d'occurrences d'un n-uplet dans une relation donnée, pour informer de la présence ou de l'absence d'un n-uplet ou encore pour signaler le degré de sécurité nécessaire pour obtenir l'accès à un certain n-uplet. Les annotations permettent donc de tracer la provenance des différents n-uplets. Lors d'une requête, elles se propagent et subissent un certain nombre d'opérations.

	Cancer	Gène muté	Annot
1	Sein	BRCA1	a
2	Sein	BRCA2	b
3	Sang	RUNX1	c

TABLE 1 – NCBIGene

	Cancer	Gène muté	Annot
1	Sein	BRCA2	d
2	Foie	HULC	e

TABLE 2 – Refseq

	Cancer	Gène muté	Provenance
1	Sein	BRCA1	a
2	Sein	BRCA2	b $+_K$ d
3	Sang	RUNX1	c
4	Foie	HULC	e

TABLE 3 – NCBIGene ∪ Refseq

Cancer	Gène muté	Annot
Sein	BRCA1	a
Sein	BRCA2	b' = b+1
Sang	RUNX1	c

TABLE 4 – NCBIGene modifiée

	Cancer	Gène muté	Provenance
1	Sein	BRCA1	a
2	Sein	BRCA2	b' $+_K$ d
3	Sang	RUNX1	c
4	Foie	HULC	e

TABLE 5 – NCBIGene modifiée ∪ Refseq

Par exemple, prenons deux relations (TABLEs 1 et 2) représentant les patients atteints d'un type de cancer et présentant une certaine mutation dans leur génôme. Ces relations sont respectivement extraites des bases NBCIgene [19] et Refseq [22], toutes deux majeures pour les données de biologie moléculaire. Elles sont annotées par le nombre de patients atteints d'un certain cancer et ayant une certaine mutation. Nous souhaitons effectuer l'union des deux relations et observer les propagations des annotations. Les n-uplets 1 et 3 de la relation résultante (représentée dans la TABLE 3) proviennent de la relation NCBIGene uniquement et leurs annotations sont donc les mêmes que celles de la TABLE 1. De même, les annotations du n-uplet 4, qui provient de la relation Refseq, sont les mêmes que le n-uplet 2 de cette dernière. Enfin, le n-uplet 2 de la TABLE 3 provient des deux relations : les annotations sont donc une combinaison des annotations des deux relations, en l'occurrence l'addition (voir plus loin).

Les annotations permettent lors d'une modification d'un n-uplet de connaître l'impact de cette modification pour le résultat d'une requête. Dans la TABLE 1, si nous ajoutons un nouveau

patient qui a le cancer du sein et la mutation BRCA2, nous obtenons la Table 4[1]. Lors de la requête effectuant l'union des deux tables, nous pouvons ainsi savoir que seul le deuxième n-uplet a été modifiée par l'ajout. En effet, le premier n-uplet de la Table 5 provient de la première ligne de la Table 1 et son annotation n'a pas été modifiée. Au contraire, le deuxième n-uplet de la Table 5 provient de la deuxième ligne de la Table 1 qui a été modifiée : b est donc remplacé par $b' = b + 1$.

Nous remarquons lors de l'union de deux n-uplets, l'utilisation de l'opérateur "+" au niveau des annotations. Dans le cas où le n-uplet est absent dans l'une des deux opérandes, le neutre "0" est utilisé, ce qui explique que l'annotation est inchangée. En outre, les opérateurs de l'algèbre relationnelle vérifiant des propriétés d'associativité, commutativité et de distributivité, il doit en aller de même pour les opérations sur les annotations : elles sont donc plongées *a minima* dans un monoïde commutatif.

2.2 Notations et définitions préliminaires

En pratique, pour capturer la richesse de l'algèbre relationnelle, les annotations sont plongées dans la structure algébrique unificatrice de semi-anneau commutatif [16].

Nous rappelons les définitions d'un monoïde, de la commutativité et d'un semi-anneau.

Définition 1 (Monoïde)**.** $(M, +, 0)$ est un monoïde si :

Loi interne $+$: $\forall x, y \in M^2, x + y \in M$

Associativité de $+$: $\forall x, y, z \in M^3, x + (y + z) = (x + y) + z$

0 **est neutre pour** $+$: $\forall x \in M, x + 0 = 0 + x = x$

Définition 2 (Commutativité)**.** Soit un ensemble M muni d'une loi interne $+$. La loi $+$ est commutative si : $\forall x, y \in M^2, x + y = y + x$.

Définition 3 (Semi-anneau)**.** $(K, 0, 1, +, \times)$ est un semi-anneau si :

1. $0 \neq 1$
2. $(K, +, 0)$ est un monoïde commutatif
3. $(K, \times, 1)$ est un monoïde
4. $\times$ est distributif par rapport à $+$: $\forall x, y, z \in K^3, x \times (y + z) = (x \times y) + (x \times z)$ et $\forall x, y, z \in K^3, (y + z) \times x = (y \times x) + (z \times y)$
5. 0 est absorbant pour le produit : $\forall x \in K, x \times 0 = 0 \times x = 0$

Lors d'une requête, les annotations subissent une combinaison d'opérations du semi-anneau commutatif K dans lequel elles sont plongées, combinaison reflétant les opérateurs de l'algèbre utilisés.

La notion usuelle de relation de l'algèbre relationnelle est un multi-ensemble de n-uplets, que l'on peut voir comme une fonction des n-uplets vers $\mathbb{N}$, indiquant le nombre d'occurrences de chaque n-uplet dans cette relation. L'idée de [16] est d'étendre les relations à des K-relations, plongeant ainsi à chaque n-uplet un élément de K.

Définition 4 (K-relations)**.** Soit $(K, 0, 1, +, *)$ un semi-anneau commutatif pour $*$ et r une relation. Une K-relation est une fonction de l'ensemble des n-uplets de r vers un élément de K.

1. Nous avons l'équivalence suivante sur les tables : une table ayant deux entrées avec le n-uplet t, l'une annoté avec a_1, l'autre avec a_2, est équivalente à une table identique à la première mais dans laquelle t a une seule entrée, annotée $a_1 +_K a_2$.

Nous noterons `annot(r,t)`, une fonction qui pour une relation r et un n-uplet t associe un élément de K. Nous nommons cet élément, l'annotation de t dans K.

Il est important de remarquer que nous nous plaçons dans le modèle usuel l'algèbre relationnelle, qui est fini. Les relations (tables) sont donc finies. En conséquence, dans le cadre des K-relations, cela revient à dire que la fonction `annot` doit être nulle presque partout, c'est-à-dire que l'ensemble des tuples t d'une relation r tels que `annot(r,t)` $\neq 0$ est fini.

Nous notons K-Algèbre, l'algèbre relationnelle étendue aux annotations plongées dans un semi-anneau commutatif K.

Nous présentons ici la syntaxe des requêtes utilisée et la sémantique de la K-Algèbre proposée par [16] avec quelques adaptations les rendant compatible avec les formalisations sur lesquelles nous nous basons dans la section suivante.

Nous utiliserons le terme *sorte* pour représenter l'ensemble des attributs d'une relation. Par exemple, la sorte de la relation de la Table 1 est {**Cancer**, **Gène muté**}.

Définition 5 (Syntaxe des requêtes)**.** La syntaxe des requêtes q est construite récursivement de la façon suivante :

$$q ::= \epsilon_{<>} \mid r \mid q \cup q \mid q \bowtie q \mid \pi(s, q) \mid \sigma(f, q)$$

où $\epsilon_{<>}$ est une relation de sorte vide et qui contient un n-uplet vide, r est une relation, l'opérateur $\cup$ correspond à l'union ensembliste, l'opérateur $\bowtie$ correspond à la jointure naturelle, l'opérateur π correspond à la projection, ici, sur un sous-ensemble s de la sorte de q, l'opérateur σ est la selection ici selon une formule f de la logique du premier ordre.

Nous notons $proj_s(t)$ la projection sur la sorte s du n-uplet t. Par exemple, la projection du n-uplet t={Sein ; BRCA1} sur la sorte contenant un seul attribut s={Cancer} nous donne le n-uplet $proj_s(t)$={Sein}.

Définition 6 (Sémantique)**.** Soit $(K, 0, 1, +, *)$ un semi-anneau commutatif. La sémantique $[\![q]\!]$ d'une requête q est une K-relation définie comme suit :

- $[\![\epsilon_{<>}]\!] = t \mapsto \begin{cases} 1 & \text{si } t =<> \\ 0 & \text{sinon} \end{cases}$
- $[\![r]\!] = t \mapsto$ `annot (r, t)`
- $[\![q_1 \cup q_2]\!] = t \mapsto [\![q_1]\!](t) + [\![q_2]\!](t)$
- $[\![q_1 \bowtie q_2]\!] = t \mapsto [\![q_1]\!](t_1) * [\![q_2]\!](t_2)$
  où les t_i correspondent aux projections de t sur la sorte des q_i associées.
- $[\![\pi(s, q)]\!] = t \mapsto \sum_{proj_s(t)=proj_s(t') \wedge [\![q]\!](t') \neq 0} [\![q]\!](t')$
- $[\![\sigma(f, q)]\!] = t \mapsto \begin{cases} [\![q]\!](t) & \text{si } eval(f, t) = \top \\ 0 & \text{sinon} \end{cases}$ où $eval$ évalue la formule logique f pour le n-uplet t

Dans le cas de la projection π, la restriction $[\![q]\!](t') \neq 0$ permet d'avoir une somme finie. En effet, comme `annot` est nulle presque partout, sa généralisation à toute requête l'est aussi.

Pour s'assurer de la correction de la sémantique de la K-algèbre relationnelle, notre contribution principale est de formaliser en Coq son adéquation avec l'algèbre relationnelle classique dans le cas de $K = \mathbb{N}$. Pour cela, nous commençons par formaliser la structure mathématique régissant les annotations -les semi-anneaux commutatifs- et certaines instances de ces semi-anneaux commutatifs.

3 Semi-anneaux de provenance en Coq

Dans cette section, nous présentons la formalisation des semi-anneaux commutatifs en Coq et nous vérifions plusieurs instances de semi-anneaux utilisées pour la provenance : les entiers naturels, les booléens et les entiers tropicaux.

3.1 Formalisation des semi-anneaux

La formalisation s'inspire et se place dans la continuation de la bibliothèque Coccinelle [10], sur laquelle est basée l'algèbre relationnelle sans annotations que nous utilisons [4]. Celle-ci définit des structures algébriques sous forme de `Record` où chaque `Record` est paramétré par le type des éléments de la structure et un ordre total sur ce type. Par exemple, notre structure de base, le monoïde commutatif, est représenté par le type Coq ci-dessous.

```
Record CM (M : Type) (zero : M) (plus : M → M → M) (TM : total_order M) : Type := ...
```

Dans cette définition, le paramètre `TM : total_order M` munit `M` d'un ordre total, avec une fonction `compare : M → M → comparison`, la relation d'équivalence `eq : M → M → Prop` associée, et les axiomes usuels d'une relation d'ordre. Il est usuel d'utiliser en Coq une équalité décidable car nous nous plaçons dans le cadre de la logique intuitionniste. Nous supposons une propriété un peu plus forte (avoir un ordre total) mais en pratique, cette propriété est toujours vérifiée. L'ordre total permet de faciliter la formalisation en Coq.

La définition du `Record` énonce les axiomes du monoïde commutatif :

```
Record CM (M : Type) (zero : M) (plus : M → M → M) (TM : total_order M) : Type :=
mk_CM
  {
    plus_assocMC : ∀ a1 a2 a3, eq (plus a1 (plus a2 a3)) (plus (plus a1 a2) a3); (*+ associatif*)
    plus_zero_leftMC : ∀ a, eq (plus zero a) a; (*0 neutre de + a gauche*)
    plus_commMC : ∀ a1 a2, eq (plus a1 a2) (plus a2 a1);(*+ commutatif*)
    plus_compat_eq_MC : ∀ a1 a2 b1 b2, eq a1 a2→  eq b1 b2 → eq (plus a1 b1) (plus a2 b2);
        (*+ compatible avec l'egalite*)
  }.
```

En utilisant cette structure de monoïde, nous pouvons définir la structure de semi-anneau commutatif :

```
Record CSR (R : Type) (zero : R) (one : R) (plus : R → R → R) (mult : R → R → R) (TR : total_order R) : Type :=
mk_CS
  {
    one_zero_SR : not (eq zero one); (*0 different de 1*)
    isMC_plus_SR : CM zero plus TR; (*(A, 0, +, TR) est un monoide commutatif*)
    isMC_mult_SR : CM one mult TR; (*(A, 1, x, TR) est un monoide commutatif*)
    mult_distr_plusSR : ∀ a1 a2 b, eq (mult b (plus a1 a2)) (plus (mult b a1) (mult b a2)); (*x distributif sur +*)
    mult_zero_absorb_leftSR : ∀ a, eq (mult a zero) zero (*0 absorbant pour x a droite*)
  }.
```

La propriété de commutativité permet d'éliminer certaines règles redondantes comme $0 \times a = 0$.

Dans les sous-sections suivantes, nous instancions cette structure de semi-anneau commutatif avec des objets mathématiques fréquemment utilisés en provenance.

3.2 Exemples de semi-anneaux commutatifs

Nous donnons dans cette sous-partie quelques exemples de semi-anneaux de provenance, et leur formalisation.

3.2.1 Entiers naturels

Annoter les données avec les entiers naturels est un moyen concis de donner leur nombre d'occurrences dans une table. Nous verrons par la suite que cela permet de relier la notion de provenance à l'interprétation usuelle des bases de données.

Dans une démarche d'efficacité d'exécution, nous avons choisi la représentation de la librairie standard `N`. L'ordre choisi est l'ordre naturel sur les entiers naturels.

3.2.2 Booléens

Les booléens sont employés pour signifier la présence ou l'absence d'un n-uplet dans une base de données selon si ce n-uplet satisfait un prédicat. C'est l'exemple le plus simple possible pour les annotations. Il est peu utile en pratique, mais des généralisations peuvent être très intéressantes, comme par exemple la logique à trois valeurs pour représenter l'incertitude de la présence ou non d'un n-uplet.

Pour les booléens, quatre choix s'offrent à nous. D'une part, le choix de 1 ($\top$ ou $\bot$) impose le 0 ($\neg 1$) et la multiplication ($\wedge$ ou $\vee$). Dans tous les cas il reste deux choix possibles pour l'addition. Le tableau suivant résume les différentes possibilités.

1	**0**	$\times$	$+$
$\top$	$\bot$	$\wedge$	$\vee$
			$\oplus$
$\bot$	$\top$	$\vee$	$\wedge$
			$\iff$

Les booléens ne comportent pas d'ordre naturel entre les deux éléments $\bot$ et $\top$. Nous avons donc encore un degré de liberté supplémentaire pour enrichir cette structure avec un ordre total : soit $\top < \bot$ soit $\bot < \top$. Les deux ordres possibles étant symétriques, le choix de l'un par préférence à l'autre n'importe pas.

3.2.3 Entiers tropicaux

La troisième instance, beaucoup plus riche, est celle des entiers tropicaux. Les entiers tropicaux peuvent représenter par exemple le coût d'obtention d'une donnée, ou en inversant l'ordre naturel sur les entiers, des niveaux de confidentialité.

Nous notons $\mathbb{N}^\infty = \mathbb{N} \cup \{+\infty\}$ et rappelons la définition des entiers tropicaux :

Définition 7 (Semi-anneau des entiers tropicaux)**.** Le semi-anneau des entiers tropicaux est le semi-anneau $(\mathbb{N}^\infty, \infty, 0_\infty, min_\infty, +_\infty)$ tel que :

— $\forall a, b \in \mathbb{N}^2, min_\infty(a, b) = min(a, b)$
— $\forall a \in \mathbb{N}^\infty, min_\infty(a, \infty) = min_\infty(\infty, a) = a$
— $\forall a, b \in \mathbb{N}^2, a +_\infty b = a + b$
— $\forall a \in \mathbb{N}^\infty, a +_\infty \infty = \infty +_\infty a = \infty$

Afin d'être le plus général possible, plutôt que de construire uniquement le semi-anneau des entiers tropicaux, nous proposons de construire une formalisation générique d'un monoïde enrichi de l'infini, du minimum et d'une règle de compatibilité de l'addition, puis de montrer que cette extension correspond à un semi-anneau commutatif.

Définition 8 (Compatibilité avec l'addition à gauche)**.** $\forall a_1, a_2, b \in M^3$, si $a_1 < a_2$ alors on a soit : $b + a_1 = b + a_2$, soit : $b + a_1 < b + a_2$.

Nous formalisons les monoïdes-compat, c'est-à-dire les monoïdes ayant cette propriété, de la façon suivante :

```
Record CMcompat (M : Type) (zero : M) (plus : M → M → M) (TM : total_order M) : Type :=
mk_CMc
{
  isCM : CM zero plus TM; (*monoide*)
  equiv_plus_compat_CM_distr_lt : ∀ a1 a2 b, (*compatibilite*)
      lt TM a1 a2 → (eq TM (plus b a1) (plus b a2) ⋁ lt TM (plus b a1) (plus b a2));
}.
```

En utilisant la structure de monoïde muni d'un ordre total, nous pouvons définir la structure de semi-anneau étendu avec l'infini et le minimum :

```
Inductive MI := (* Ajout d'un element infini *)
| Infinity : MI
| EltM : M → MI.

Definition compMI a1 a2 := (*Extension de la comparaison avec l'infini : borne superieure*)
  match a1 with
  | Infinity ⇒
    match a2 with
    | Infinity ⇒ Eq
    | EltM b2 ⇒ Gt
    end
  | EltM b1 ⇒
    match a2 with
    | Infinity ⇒ Lt
    | EltM b2 ⇒ (t_compare TM) b1 b2
    end
  end.

Definition mulMI a1 a2 := (*Extension de l'addition avec l'infini (cela sera la multiplication du semi−anneau)*)
  match a1 with
  | Infinity ⇒ Infinity
  | EltM b1 ⇒
    match a2 with
    | Infinity ⇒ Infinity
    | EltM b2 ⇒ EltM (plus b1 b2)
    end
  end.

Definition minMI a1 a2 := (*Minimum (cela sera l'addition du semi−anneau)*)
  match compMI a1 a2 with
  |Gt ⇒ a2
  |_ ⇒ a1
  end.
```

Nous prouvons enfin le résultat suivant :

Théorème 1 (Monoïde-compat étendu avec l'infini)**.** Un monoïde-compat présentant un ordre total, peut être étendu avec l'infini et le minimum ; cette extension forme un semi-anneau commutatif.

Cette proposition nous permet d'ajouter de manière systématique, un infini et un minimum à une structure simple. Ainsi, pour prouver le caractère de semi-anneau commutatif des entiers tropicaux, il suffit donc de montrer que la structure des entiers naturels munie de l'ordre naturel est un monoïde-compat.

4 K-Algèbre en Coq

Nous souhaitons formaliser la sémantique de l'algèbre relationnelle annotée définie dans la section 2.

4.1 Contexte : formalisation de l'algèbre relationnelle

Nous nous inscrivons dans la continuité de l'article [4] qui formalise la syntaxe et la sémantique de l'algèbre relationnelle des bases de données sans annotations. Nous utilisons donc le type inductif `query` proposé pour la syntaxe des requêtes.

```
Inductive query : Type :=
  | Q_Empty_Tuple : query (*Relation contenant un n-uplet vide*)
  | Q_Table : relname → query (*Relation identifiee par un nom*)
  | Q_union : query → query → query (*Union*)
  | Q_Join : query → query → query (*Jointure naturelle*)
  | Q_Pi : list select → query → query (*Projection*)
  | Q_Sigma : q_formula → query → query (*Selection*)
```

Dans cette formalisation, `relname` est un type abstrait représentant l'ensemble des noms de relations. Une preuve de concept instancie ce type par le type `string` des chaînes de caractères, comme usuel en SQL.

La sémantique de l'algèbre relationnelle proposée dans [4] est définie par une fonction récursive `eval_query_rel` qui à chaque requête associe une relation. Les relations sont formalisées par des multi-ensembles de n-uplets. Pour cela, nous utilisons la bibliothèque des multiensembles finis `Febag`, qui se place dans le prolongement de Coccinelle [10].

```
Fixpoint eval_query_rel q : Febag.bag :=
  match q with
  | Q_Empty_Tuple ⇒ Febag.singleton empty_tuple (*multi-ensemble contenant un n-uplet vide*)
  | Q_Table r ⇒ instance_rel r
  | Q_union q1 q2 ⇒
    if sort q1 =?^set sort q2
    then Febag.union (eval_query_rel q1) (eval_query_rel q2) (*union de deux multi-ensembles*)
    else Febag.empty
  | Q_Join q1 q2 ⇒ natural_join_bag (eval_query_rel q1) (eval_query_rel q2)
                   (*Resultats de la jointure naturelle sur les multi-ensembles*)
  | Q_Pi s q ⇒ Febag.map (fun t ⇒ projection t s) (eval_query_rel q)
  | Q_Sigma f q ⇒ Febag.filter (fun t ⇒ eval_q_formula t f) (eval_query_rel q)
  end.
```

La fonction `instance_rel` associe au nom (unique) de la relation, le multi-ensemble des n-uplets contenus dans la relation.

La fonction `Febag.map` applique une fonction sur un multi-ensemble. Dans notre cas, elle applique la fonction `projection` qui renvoie la projection de t sur la sorte s. La fonction `sort` permet de donner les attributs pour une requête donnée. La fonction `Febag.filter` permet de ne conserver que les éléments d'un multi-ensemble qui satisfont un prédicat. Ici notre prédicat évalue une formule logique f sur un n-uplet t à travers la fonction `eval_q_formula`.

Dans le cas de l'union, si les sortes (les ensembles des attributs) sont bien les mêmes pour les deux sous-requêtes alors il est renvoyé l'union des deux multi-ensembles (en conservant les doublons éventuels). Si cela n'est pas le cas, il est renvoyé le multi-ensemble vide.

La formalisation de nos K-relations se base sur la syntaxe des requêtes précédentes.

4.2 Formalisation des K-relations

Dans cette sous-section, nous formalisons les K-relations et l'algèbre relationnelle pour la provenance.

Une K-relation est formalisée par le type `tuple → K` où `tuple` correspond à l'ensemble des n-uplets possibles et K est un semi-anneau commutatif. K est donc de type `CSR`, et nous noterons pour simplifier ses champs `zero, one, ...` et ainsi de suite. A partir de cela, nous définissons la sémantique de l'algèbre relationnelle sous la forme d'une fonction récursive `eval_query_prov` qui prend une requête et renvoie la K-relation correspondante.

```
Fixpoint eval_query_prov q : (tuple → K) :=
  match q with
  | Q_Empty_Tuple⇒ fun t ⇒ if eq_bool t empty_tuple then one else zero
  | Q_Table r ⇒ instance_prov r
  | Q_union q1 q2 ⇒
    if sort q1 =?^set sort q2
    then fun t ⇒ plus ((eval_query_prov q1) t) ((eval_query_prov q2) t)
    else fun t ⇒ zero
  | Q_Join q1 q2 ⇒
    fun t ⇒
      mul
        (mul
          ((eval_query_prov q1) (mk_tuple T (sort q1) (dot t))) (*annotation de la projection du n−uplet sur les attributs
                de q1*)
          ((eval_query_prov q2) (mk_tuple T (sort q2) (dot t)))) (*annotation de la projection du n−uplet sur les attributs
                de q1*)
        (if support t =?^set (sort q1 ∪ sort q2) then one else zero)
  | Q_Pi s q ⇒
  fun t' ⇒ finite_sum
              (List.map (eval_query_prov q)
                  (List.filter (fun t ⇒ eq_bool t' (projection t s) && negb (eq_bool ((eval_query_prov q) t) zero))
                      (content_of_query q)))
  | Q_Sigma f q ⇒ fun t ⇒ mul (
      match (eval_q_formula t f) with
      | true ⇒ one
      | false ⇒ zero
      end) ((eval_query_prov q) t)
    end
  end
```

Dans le cas du n-uplet vide, la fonction renvoyée vérifie si le n-uplet est équivalent au n-uplet vide et donne l'annotation un si c'est le cas et zéro sinon. Pour la table, la fonction `instance_prov` associe au nom de la relation, la K-relation correspondante. Dans le cas de l'union, il est vérifié que les sortes sont bien égales dans un premier temps. Si cela est le cas, il est retourné la fonction qui effectue l'addition des annotations des deux relations pour un n-uplet donné. Si cela n'est pas le cas, il est renvoyé la fonction constante égale à zéro. Pour la jointure naturelle, il faut que la fonction multiplie l'annotation de la projection du n-uplet sur les attributs de la première sous-requête avec l'annotation de la projection du n-uplet sur les attributs de la deuxième sous-requête mais aussi vérifie si le support du n-uplet est égal à l'union des sortes des deux requêtes. Dans le cas de la projection, il est associé à chaque n-uplet t la somme des annotations des n-uplets qui correspondent à la projection de t sur une sélection d'attributs s. La somme s'effectue à l'aide de la fonction `finite_sum` qui prend une liste d'éléments de K. Pour parcourir tous les n-uplets possibles dont la projection sur s est équivalente à t, nous avons besoin d'introduire l'ensemble des n-uplets contenus dans la sous-requête qui est formalisée par la fonction `content_of_query`. Cet ensemble ne contient qu'une seule occurrence de chaque n-uplet. Pour récupérer les bons n-uplets, la fonction `eval_query_prov` filtre les éléments de `content_of_query` dont la projection équivaut à t et vérifie que l'annotation de ces éléments n'est pas zéro. Il reste donc

à sommer les annotations de ces éléments. Enfin, la sélection renvoie une fonction qui dans un premier temps évalue la formule logique f sur un n-uplet avec la fonction `eval_q_formula`. Si le résultat est vrai alors nous renvoyons l'évaluation de la sous requête sur le n-uplet et sinon nous renvoyons zéro.

4.3 Propriétés générales

Dans cette sous-section, nous proposons quelques propriétés générales qui découlent de nos définitions.

Si nous supposons que l'annotation d'un certain n-uplet pour une requête n'est pas égale à zéro, nous pouvons prouver que ce n-uplet apparait bien dans la relation.

```
Lemma mem_eval_content : ∀ (q: query) (t: tuple),
    not (eq ((eval_query_prov q) t) zero) →
    mem_bool t (content_of_query q) = true.
```

Par ailleurs, nous prouvons également que si un n-uplet est dans une relation, le support de ce n-uplet est égal à la sorte de la requête.

```
Lemma in_content_support_sort: ∀ (q:query) (t:tuple),
      In t (content_of_query q) →
      support t =set= sort q.
```

Avec les deux lemmes précédents, nous pouvons enfin prouver que si l'annotation pour un certain n-uplet n'est pas égale à zéro alors le support de cet n-uplet est égal à la sorte de la requête.

```
Lemma not_zeroK_eq_support_sort : ∀ (q:query) (t:tuple),
    not (eq ((eval_query_prov q) t) zero) →
    support t =set= sort q.
```

5 Théorème d'adéquation

Nous pouvons remarquer qu'avoir des multi-ensembles de n-uplets et une fonction comptant le nombre d'occurrences des n-uplets sont deux façons analogues de définir des multi-ensembles.

Autrement dit, si nous prenons les relations et les K-relations correspondantes ayant pour annotation le nombre d'occurrences de chaque n-uplet, nous devons avoir une égalité entre le nombre d'occurrences d'un n-uplet lors de l'évaluation d'une requête q et l'annotation associée à ce n-uplet pour cette même requête q. Le semi-anneau approprié pour ces K-relations est celui des entiers naturels.

Pour avoir cette égalité, il est nécessaire de faire coïncider le cas de l'opérateur `Q_table`, c'est-à-dire que pour une relation nommée r, le nombre d'occurrences d'un n-uplet t dans le multi-ensemble `instance_rel r` doit être égal à l'annotation `instance_prov r t`. Nous posons donc l'hypothèse suivante :

```
Hypothesis instance_prov_nb_occ : ∀ (r : relname), (t : tuple),
            instance_prov r t = nb_occ t (instance_rel r).
```

La fonction `nb_occ` compte le nombre d'occurrences d'un n-uplet dans un multi-ensemble.

Pour prouver l'égalité pour toutes les requêtes, nous effectuons une induction sur la structure des requêtes. Le cas de la projection est le plus délicat à prouver. Cela nécessite de prouver que la somme finie de chaque annotation pour chaque n-uplet obtenue par la projection d'un n-uplet t est égale à une somme de chaque occurrence de chaque n-uplet obtenue par la projection d'un n-uplet t.

Nous introduisons une fonction auxiliaire pour prouver cette propriété.

```
Fixpoint map_reduce (e:tuple) (f : tuple → tuple) (g : tuple → N) (l : list (tuple)) :=
match l with
  | nil ⇒ 0
  | cons a l' ⇒ (if eq_bool e (f a) then 1 else 0) ∗ (g a)) + (map_reduce e f g l')
end.
```

Pour chaque n-uplet `e`, `map_reduce` parcourt la liste `l` et pour chaque élément `a` de `l`, elle vérifie s'il y a équivalence entre `e` et `f a` et si c'est le cas, ajoute `(g a)`.

Cette fonction nous permet de faire le pont entre l'évaluation dans l'algèbre relationnelle classique et dans la $\mathbb{N}$-algèbre. En effet, dans un premier temps, nous relions cette fonction à l'évaluation d'une part dans l'algèbre, d'autre part dans la $\mathbb{N}$-algèbre :

```
Lemma nb_occ_rel_map_reduce : ∀ (e:tuple),
    nb_occ e (Febag.map (fun t : tuple ⇒ projection t s) (eval_query_rel q)) =
    map_reduce e (fun t : tuple ⇒ projection t s) (fun _: tuple ⇒ 1) (Febag.elements (eval_query_rel q)).
```

et

```
Lemma finite_sum_map_reduce : ∀ (e:tuple),
  finite_sum
    (map (eval_query_prov_N q)
       (filter
          (fun t : tuple ⇒ eq_bool e (projection t s) && negb (eq_bool (eval_query_prov_N q t) 0))
                (content_of_query q))) =
  map_reduce e (fun t : tuple ⇒ projection t s)
    (fun t : tuple ⇒ nb_occ t (Febag.elements (eval_query_rel q)))
    (filter (fun x : tuple ⇒ negb (eq_bool (eval_query_prov_N q x) 0)) (content_of_query q)).
```

Dans un deuxième temps, nous voulons donc prouver que :

```
map_reduce e (fun t : tuple ⇒ projection t s) (fun _: tuple ⇒ 1) (Febag.elements (eval_query_rel q)) =
map_reduce e (fun t : tuple ⇒ projection t s)
    (fun t : tuple ⇒ nb_occ t (Febag.elements (eval_query_rel q)))
    (filter (fun x : tuple ⇒ negb (eq_bool (eval_query_prov_N q x) 0)) (content_of_query q))
```

Nous remarquons que la fonction `f` est la même dans les deux cas. Les deux listes correspondent aux n-uplets contenus dans la relation. Dans le cas des multi-ensembles, les n-uplets de la relation sont dans le multi-ensemble de `eval_query_rel q`. Pour les annotations, ces n-uplets correspondent aux n-uplets contenus dans `content_of_query` qui possèdent une annotation non nulle. Comme remarqué précédemment, `content_of_query` ne contient qu'une seule occurrence de chaque élément alors que le multi-ensemble peut en contenir plusieurs. La difficulté réside donc dans le fait que si nous avons un n-uplet `a` qui vérifie que `eq_bool e (f a)`, pour les multi-ensembles chaque occurrence est ajoutée, une à une, par la fonction constante égale à 1 alors que pour les annotations, le nombre d'occurrences de `a` est ajouté au plus une seule fois.

Pour cela, nous avons prouvé le lemme suivant :

```
Lemma equal_map_reduce_prov_rel : ∀ (e:tuple) (f:tuple→ tuple) (l1:list tuple) (l2 :list tuple),
    (∀ a1 a2, eq a1 a2 →
                    eq (f a1) (f a2)) →
    (∀ t, nb_occ t l1 =0 ⟷ nb_occ t l2 = 0) →
    (∀ t,
       nb_occ t l2 = 0 ⋁ nb_occ t l2 = 1) →
    map_reduce e f (fun t ⇒ nb_occ t l1) l2 =
    map_reduce e f (fun t ⇒ 1) l1.
```

Les hypothèses proposées supposent que la fonction projection est compatible avec l'équivalence, que si un n-uplet `t` n'est pas dans l'une des listes, il n'est pas dans l'autre ainsi que la liste `l2` a au plus une occurrence de chaque élément. La première hypothèse est raisonnable car la projection de deux n-uplets équivalents doit donner deux projections équivalentes. Les deux autres hypothèses découlent des remarques précédentes.

Avec ce lemme, nous avons obtenu l'égalité dans le cas de la projection. Les autres cas présentaient moins de subtilité. Nous avons donc prouvé notre théorème fondamental : [2]

```
Theorem K_relations_extend_relational_algebra :
  ∀(q:query),
    ∀(t:tuple), (eval_query_prov q) t = nb_occ t (eval_query_rel q).
```

ce qui se traduit par :

Théorème 2. Les K-relations sont une extension de l'algèbre relationnelle.

Autrement dit, en instanciant K par le semi-anneau commutatif des entiers naturels, nous retrouvons la sémantique de l'algèbre relationnelle.

6 Etat de l'art

Sur le versant de la formalisation de modèles, langages ou systèmes centrés données, des contributions ont vu le jour. Ainsi, la première tentative de formalisation, en Agda [24], de l'algèbre relationnelle est proposée par [13, 12] tandis que la première formalisation, complète, du modèle relationnel est proposée par [4] où le modèle de données, l'algèbre, les requêtes «tableaux», la procédure de semi-décision «chase» et les contraintes d'intégrité sont formalisés.

La toute première tentative de vérifier, en Coq, un SGBDR est présentée dans [20]. Des propositions de sémantiques mécanisées pour SQL sont présentées dans [2, 3]. Une formalisation d'un moteur de requêtes SQL est fournie dans [5].

À notre connaissance, il n'existe pas de travaux sur la formalisation au moyen d'assistants à la preuve de la provenance des données. Une formalisation des K-relations est donnée dans l'article [8] proposant un outil pour décider de l'équivalence de deux requêtes SQL. À cette fin, HottSQL, une sémantique pour un fragment de SQL est définie. Cette sémantique se base sur la notion de K-relation. Toutefois, la modélisation des K-relations proposée *relaxe la contrainte de finitude* de celles ci. Ainsi, cette propositon peut conduire à manipuler des instances de K-relations potentiellement infinies. En outre, la sémantique présentée n'est pas *exécutable*. Notre approche permet d'avoir une sémantique exécutable, ce qui offre la possibilité d'extraire du code, d'exécuter des tests, etc.

Sur le versant de la formalisation de structures algébriques, de nombreuses propositions ont été développées en Coq, la plus élaborée étant Mathematical Components [23]. Notre formalisation des semi-anneaux repose sur la bibliothèque Coccinelle [10] pour deux raisons : la compatibilité avec la formalisation de l'algèbre relationnelle sur laquelle repose ce travail, et, de nouveau, la volonté de fournir une sémantique exécutable. Sur ce deuxième aspect, nous envisageons de nous lier à Mathematical Components via le mécanisme de raffinements proposé par Cohen *et al.* [9].

7 Conclusions et perspectives

Nous avons présenté les bases pour la formalisation de la provenance des données. En nous basant sur des formalisations existantes de l'algèbre relationnelle d'une part, et des structures algébriques usuelles d'autre part, nous avons développé le modèle de provenance basé sur les K-relations. Notre avons enfin prouvé le théorème fondamental montrant l'adéquation de cette

2. Dans la formalisation disponible en ligne, les requêtes contiennent des agrégats qui ne sont pas encore traités dans notre formalisation de la provenance.

généralisation avec l'algèbre relationnelle usuelle. Notre formalisation offre un niveau d'abstraction très élevé, et est notamment paramétrique en le semi-anneau de provenance. Elle est développée dans l'assistant de preuve Coq.

Notre prochaine étape est d'étendre les K-algèbres au cas des agrégats comme proposé dans [1], et ainsi avoir une sémantique pour les annotations pour les agrégats mécanisée et exécutable en Coq. La combinaison dans le cas des agrégats repose sur la notion de semi-module, qui permet d'agréger des annotations (comme les agrégats agrègent les données).

Nous souhaitons lier ce travail à la formalisation du langage de requêtes SQL et à sa traduction certifiée vers l'algèbre relationnelle [3]. Pour cela, nous voulons proposer un langage de requêtes tenant compte des annotations et les traduisant vers l'algèbre de manière correcte et complète.

À long terme, notre objectif est de certifier la provenance dans des analyses de données telles qu'utilisées en réseau, bio-informatique, etc. Notre cas d'étude sera les workflows scientifiques qui, à partir de données d'entrées et d'une suite d'instructions (pouvant contenir des boucles), génèrent de nouvelles données ensuite analysées. Ainsi, les workflows scientifiques seront instrumentés pour fournir une trace, dont l'outil certifié en Coq pourra garantir la reproductibilité : ce processus de certification *a posteriori* reposera sur une approche sceptique, et sera donc indépendante des outils apparaissant dans les workflows [18]. Pour mener à bien la vérification de ces traces, il faudra définir un langage formel de spécification de workflows, permettant d'établir des propriétés sur ces derniers et de montrer l'équivalence de (parties de) workflows, comme requis dans [6] pour garantir la reproductibilité. L'un des enjeux de cette phase sera d'assurer un passage à l'échelle au niveau du volume des données et des requêtes posées. Nous travaillons actuellement sur la constitution de cas d'utilisations réels pour évaluer plus précisément les capacités de notre solution sur ces aspects.

Remerciements : Nous remercions les rapporteurs anonymes pour leurs remarques pertinentes.

Références

[1] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 153–164. ACM, 2011.

[2] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1555–1569, 2017.

[3] Véronique Benzaken and Évelyne Contejean. A coq mechanised formal semantics for realistic SQL queries : formally reconciling SQL and bag relational algebra. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 249–261. ACM, 2019.

[4] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A Coq Formalization of the Relational Data Model. In Zhong Shao, editor, *ESOP - 23rd European Symposium on Programming*, Lecture Notes in Computer Science, Grenoble, France, April 2014. Springer.

[5] Véronique Benzaken, Evelyne Contejean, Ch. Keller, and E. Martins. A coq formalisation of sql's execution engines. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC*

2018, Oxford, UK, July 9-12, 2018, Proceedings, volume 10895 of *Lecture Notes in Computer Science*, pages 88–107. Springer, 2018.

[6] Sarah Cohen Boulakia, Khalid Belhajjame, Olivier Collin, Jérôme Chopard, Christine Froidevaux, Alban Gaignard, Konrad Hinsen, Pierre Larmande, Yvan Le Bras, Frédéric Lemoine, Fabien Mareuil, Hervé Ménager, Christophe Pradal, and Christophe Blanchet. Scientific workflows for computational reproducibility in the life sciences : Status, challenges and opportunities. *Future Generation Comp. Syst.*, 75 :284–298, 2017.

[7] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where : A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings.*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[8] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL : Proving query rewrites with univalent SQL semantics. In *PLDI*. ACM, 2017.

[9] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free ! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[10] Évelyne Contejean. Coccinelle, a Coq library for rewriting. In *Types*, Torino, Italy, March 2008.

[11] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2) :179–227, June 2000.

[12] C. Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In *RelMiCS*, pages 137–148, 2003.

[13] C. Gonzalia. *Relations in Dependent Type Theory.* PhD thesis, Chalmers Göteborg University, 2006.

[14] Todd J. Green. Containment of conjunctive queries on annotated relations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 296–309. ACM, 2009.

[15] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40. ACM, 2007.

[16] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40. ACM, 2007.

[17] Todd J. Green and Val Tannen. The semiring framework for database provenance. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 93–99. ACM, 2017.

[18] J. Harrison and L. Théry. A Sceptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3) :279–294, 1998.

[19] Donna Maglott, Jim Ostell, Kim D Pruitt, and Tatiana Tatusova. Entrez gene : gene-centered information at ncbi. *Nucleic acids research*, 33(suppl_1) :D54–D58, 2005.

[20] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *ACM Int. Conf. POPL*, 2010.

[21] Laurel Orr, Dan Suciu, and Magdalena Balazinska. Probabilistic database summarization for interactive data exploration. *PVLDB*, 10(10) :1154–1165, 2017.

[22] Kim D Pruitt, Tatiana Tatusova, Garth R Brown, and Donna R Maglott. Ncbi reference sequences (refseq) : current status, new features and genome annotation policy. *Nucleic acids research*, 40(D1) :D130–D135, 2011.

[23] The Mathematical Components team. Mathematical Components. Available at https://math-comp.github.io/math-comp.

[24] The Agda Development Team. *The Agda Proof Assistant Reference Manual*, 2010.

FaCiLe en Coq : vérification formelle des listes d'intervalles

Amélie Ledein[1], Catherine Dubois[1,2]

[1] ENSIIE, Évry, France
[2] Samovar, Évry, France
{amelie.ledein, catherine.dubois}@ensiie.fr

Abstract

Lors du développement d'un solveur de contraintes, la représentation des domaines des variables est un choix de conception important car ce dernier a une forte influence sur l'efficacité du solveur. Une représentation assez courante - et judicieuse lorsque les domaines sont grands et *avec peu de trous* - consiste à stocker une séquence d'intervalles aussi larges que possible. La bibliothèque OCaml de programmation par contraintes sur les domaines finis, nommée FaCiLe (Barnier, Brisset, 1997), ainsi que de nombreux solveurs, implantent les domaines avec une telle représentation. Nous présentons dans cet article la formalisation en Coq du module de FaCiLe dédié à la création et la manipulation des domaines. Dans ce travail, l'effort a porté non seulement sur la preuve mais surtout sur la spécification des fonctions de ce module. Plus généralement, nous proposons une nouvelle implantation Coq des ensembles finis d'entiers. Notre objectif est d'utiliser le module Coq correspondant dans le solveur de contraintes formellement vérifié développé par Carlier et al, afin d'obtenir de meilleures performances pour le code extrait. Ce travail participe également à l'effort de vérification formelle des bibliothèques OCaml existantes.

1 Introduction

La programmation par contraintes est une technique permettant de résoudre des problèmes fortement combinatoires. Un problème de satisfaction de contraintes est défini à partir de variables munies chacune d'un domaine définissant l'ensemble des valeurs possibles, et de contraintes qui expriment des propriétés qui doivent être satisfaites par les variables. Un tel problème est ensuite soumis à un solveur dédié qui retournera une ou plusieurs solutions, voire toutes les solutions, ou UNSAT si le problème n'admet pas de solution. Il existe de nombreux solveurs ou bibliothèques de résolution de contraintes, comme par exemple Choco [15], Minion [6], Gecode [5], MiniZinc [3], ECLiPSe Prolog [4], SICStus Prolog [7], FaCiLe [8].

Carlier, Dubois et Gotlieb ont développé, en 2012, un solveur de contraintes binaires à domaines finis [9], appelé CoqBinFD dans la suite. Ce solveur a la particularité d'avoir été développé avec l'assistant à la preuve Coq et d'avoir été démontré correct et complet. Ainsi quand il répond UNSAT, il a été démontré qu'il n'y a effectivement pas de solution au problème soumis. Cependant, les performances actuelles de ce solveur sont loin d'être comparables à celles des autres solveurs. Outre l'introduction de techniques de résolution dédiées (comme par exemple pour la contrainte n-aire *alldifferent*), un axe d'optimisation concerne la représentation des domaines qui a une forte influence sur l'efficacité d'un solveur. Une représentation assez courante - et judicieuse lorsque les domaines sont grands et *avec peu de trous* - consiste à utiliser une séquence d'intervalles aussi larges que possible. La bibliothèque OCaml de programmation par contraintes sur les domaines finis FaCiLe [8], développée par Barnier et Brisset dans les années 90, ainsi que de nombreux solveurs, implantent les domaines avec une telle représentation.

Nous proposons de formaliser et vérifier formellement une représentation des domaines à base de listes d'intervalles et de l'intégrer au solveur CoqBinFD en nous appuyant sur l'implantation fournie dans la bibliothèque OCaml FaCiLe (http://facile.recherche.enac.fr/).

Dans cet article, une première contribution est la vérification formelle avec Coq d'une partie du module implantant les domaines de FaCiLe, complétée par quelques fonctions utiles au solveur CoqBinFD. Une deuxième contribution est la proposition d'une nouvelle implantation formellement vérifiée des ensembles finis d'entiers qui étend le développement formel précédent. Le code Coq est disponible à l'adresse suivante https://gitlab.com/finite_set_Coq/real_intervals_list.

Le plan de cet article est le suivant. Dans un premier temps, à la section 2, nous introduisons plus précisément le contexte, à savoir la définition mathématique d'un problème de satisfaction de contraintes, les grandes lignes de la résolution, ainsi qu'une rapide présentation concernant la représentation des domaines dans les solveurs actuels. La section 3 introduit la bibliothèque FaCiLe, et plus particulièrement son module dédié à la représentation de domaines entiers, et présente également sa traduction partielle en Coq. La section 4 traite plus en détails des méthodes suivies pour spécifier et prouver en Coq la correction des fonctions implantées. Dans la section 5, nous présentons différentes extractions vers OCaml et comparons les temps d'exécution. La dernière section conclut et présente quelques perspectives.

2 Contraintes sur des domaines finis

Cette section présente brièvement la notion de problème de satisfaction de contraintes à domaines finis, ainsi que les mécanismes de la résolution de ces problèmes. Les choix faits lors du développement de CoqBinFD y sont rappelés.

Un *problème de satisfaction de contraintes sur des domaines finis* (ou CSP en abrégé, pour *Constraint Satisfaction Problem*) modélise un problème à l'aide d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine donné. Formellement, un CSP est défini par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ est l' ensemble des n variables du problème,
- $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ est l'ensemble des domaines finis des n variables,
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ est un ensemble de m contraintes. Chaque contrainte C_k met en relation certaines variables de $\mathcal{X}$, ce qui restreint les valeurs que peuvent prendre simultanément chacune de ces variables. Ces contraintes peuvent être numériques (sur des réels, entiers, linéaires ou non, etc.), booléennes, sur des ensembles, etc.

Dans la suite, nous considérons des problèmes où les contraintes ne changent pas au cours de la résolution.

Selon l'objectif recherché, *résoudre un problème de satisfaction de contraintes* signifie : prouver l'existence d'une solution ou non, trouver une solution, trouver l'ensemble des solutions du problème, trouver une solution optimale par rapport à un critère (généralement minimisation ou maximisation d'une fonction de coût), prouver l'appartenance d'une valeur d'une inconnue à une solution, prouver l'appartenance d'une valeur d'une inconnue à toutes les solutions, etc. Le plus souvent, l'objectif est de trouver au moins une solution ou de prouver que le problème est UNSAT, i.e. qu'il n'a pas de solution.

Pour résoudre un CSP, la plupart des solveurs existants entrelacent 3 processus : filtrage, propagation et énumération. Le *filtrage d'une contrainte* consiste à supprimer les valeurs inconsistantes des domaines des variables de la contrainte. La consistance est ici relative à une notion

de consistance locale. Par exemple, l'*hyper-consistance* requiert que, pour chaque variable de la contrainte, pour chacune de ses valeurs possibles, il existe des valeurs (appelées *supports*) pour les autres variables telles que la contrainte soit satisfaite. Si, pour une valeur d'un domaine, cette condition n'est pas satisfaite, alors cette valeur est retirée du domaine. La *consistance de bornes* affaiblit la condition en n'imposant l'existence d'un support que pour les bornes inférieure et supérieure des domaines. Le solveur CoqBinFD s'appuie sur la *consistance d'arc* (hyper-consistance réduite à deux variables), son filtrage est prouvé complet, c'est-à-dire qu'il retire toutes les valeurs inconsistantes des domaines.

Lors du développement d'un solveur de contraintes, la représentation des domaines des variables est un choix de conception important car ce dernier a une forte influence sur l'efficacité du solveur. Une représentation assez courante - et judicieuse lorsque les domaines sont grands et *avec peu de trous* - consiste à utiliser une séquence d'intervalles aussi larges que possible. Elle est implantée par exemple dans les solveurs ECLiPSe Prolog, SICStus Prolog, Choco et FaCiLe. On trouve également d'autres représentations [16], comme les *bitvectors* (par exemple par ILOG Solver [1]), les *gap interval trees* qui stockent les valeurs qui ne sont pas dans le domaine [14] (par exemple Naxos Solver [2]) ou les ensembles creux [13] (Oscar et Castor par exemple). Le solveur CoqBinFD, quant à lui, utilise des listes triées sans doublons pour représenter les domaines.

3 FaCiLe et ses domaines

3.1 Description de la bibliothèque FaCiLe

Afin d'améliorer le type de structure utilisé dans le solveur CoqBinFD, nous avons traduit une partie d'une bibliothèque fonctionnelle de programmation par contraintes sur les domaines finis, nommée FaCiLe (*Functional Constraint Library*) [8]. Cette bibliothèque documentée[1] a été écrite en OCaml par Barnier et Brisset, notamment pour diversifier les langages de programmation avec lesquels utiliser un solveur. En effet, de très nombreux solveurs s'utilisent avec C++, Java ou Prolog. La bibliothèque FaCiLe est simple, gratuite et libre (licence LGPL), petite (moins de 5 000 lignes organisées en une douzaine de modules), d'une bonne efficacité et extensible car composée de briques de base. Elle a été inspirée notamment par ECLiPSe Prolog et ILOG Solver. Sa dernière mise à jour officielle date de décembre 2016.

FaCiLe permet de poser des contraintes sur des variables dont le domaine est un ensemble fini d'entiers ou encore un ensemble d'ensembles finis d'entiers. Nous nous intéressons uniquement au premier cas. Le module `Domain` est dédié à la représentation des domaines entiers : il offre le type abstrait `Domain.t` ainsi que de nombreuses fonctions de construction et de manipulation des domaines entiers. Un domaine est donc représenté par une liste d'intervalles eux-mêmes représentés par la donnée de leurs bornes inférieure et supérieure. L'hypothèse implicite est que la liste est triée dans l'ordre croissant des bornes et que les intervalles sont aussi grands que possible, par exemple l'ensemble {1, 2, 3, 4, 5, 7, 10, 11} sera représenté par la liste `[(1,5);(7, 7);(10, 11)]`. Ainsi la différence entre la borne inférieure d'un intervalle et la borne supérieure de l'intervalle précédent, s'il existe, est supérieure ou égale à 2. Plus précisément un domaine est un enregistrement contenant une liste d'intervalles (champ `domain`), la longueur de cette liste (champ `size`), ainsi que la valeur minimale (champ `min`) et la valeur maximale (champ `max`) de la liste d'intervalles. Lorsque le domaine est vide, les champs `min` et `max` reçoivent la valeur `min_int`.

[1] 1 600 lignes de signatures (interfaces) documentées (http://opti.recherche.enac.fr/facile/index.html.fr)

Les domaines sont construits à l'aide des fonctions suivantes (liste non exhaustive) : `Domain.empty` crée le domaine vide ; `Domain.create` crée un domaine à partir d'une liste d'entiers qui peut être non triée et contenir des doublons ; `Domain.interval` construit un domaine correspondant à un intervalle d'entiers ; `Domain.remove` retire un élément d'un domaine ; `Domain.add` ajoute un élément à un domaine. Le module offre également un grand nombre d'opérateurs ensemblistes comme l'intersection, l'union et la différence.

Les fonctions de manipulation des domaines permettent de tester l'appartenance d'un entier à un domaine (`Domain.member`), d'obtenir la longueur les valeurs minimale (`Domain.min`) et maximale (`Domain.max`) d'un domaine ou encore d'imprimer les domaines. Enfin une fonction `Domain.values` donne la liste exhaustive, et dans l'ordre croissant, des entiers contenus dans un domaine. Nous ne détaillons pas le code OCaml car il sera décrit au travers de sa traduction en Coq dans les sections suivantes.

3.2 Traduction manuelle vers Coq

L'étape de traduction du code OCaml en Coq a été réalisée manuellement, de manière quasi syntaxique. En effet, le code établi dans le module `Domain` est proche d'un code fonctionnel sans effet de bord, ce qui rend sa traduction en Coq assez simple. Parmi les fonctions traduites, une seule utilise une donnée mutable. Quelques fonctions utilisent des exceptions et des assertions. Ce caractère fonctionnel justifie notre choix d'utiliser ce module.
Afin d'assurer la traçabilité entre le code OCaml et le code Coq, les noms des fonctions n'ont pas été changés. Nous avons essayé de rester le plus proche possible du code OCaml. Cependant, quelques adaptations ont été nécessaires, notamment à cause des quelques différences existantes entre ces 2 langages : exceptions, assertions, filtrage exhaustif et terminaison des fonctions notamment. Pour plus de simplicité dans la preuve, nous avons désimbriqué les fonctions dont le code OCaml fait un usage fréquent. En ce qui concerne la terminaison, ce fut assez simple car les fonctions présentent pour la plupart une récursion structurelle ou une récursion générale dont la terminaison s'appuie sur la longueur des listes. La construction `Function` a été utilisée dans ce cas, ainsi que l'induction fonctionnelle associée lors de la démonstration de propriétés. Certaines fonctions ont donné lieu, néanmoins, à quelques modifications lors de leur traduction en Coq, pour raison d'optimisation par exemple.

Une alternative à la traduction manuelle est l'utilisation d'outils comme CoqOfOCaml [12] ou CFML [10]. Cependant, certains traits de OCaml utilisés dans cette bibliothèque ne sont pas gérés par CoqOfOCaml, ce qui empêche la traduction. De plus, cet outil introduit dans le code Coq des monades, ce qui obscurcirait inutilement le code Coq. Le code Coq obtenu avec la traduction de CFML est lui aussi éloigné du code OCaml initial, et ne facilite pas la preuve.

Finalement, 206 lignes de la bibliothèque FaCiLe définissant 29 fonctions et 2 types ont été traduites. Le code Coq correspondant compte 57 fonctions. La différence est due aux nombreuses fonctions imbriquées présentes dans le code OCaml.

4 Spécification et preuve formelles

En plus des fonctions évoquées plus haut, nous avons complété le développement avec quelques fonctions requises par le solveur CoqBinFD comme celles qui permettent d'explorer un domaine pour établir la consistance d'arc d'une contrainte. D'autre part, notre objectif étant également de fournir une implantation des ensembles finis d'entiers à l'aide de listes d'intervalles, nous avons enfin complété notre développement formel avec les fonctions manquantes spécifiées dans l'interface `FSet` de la bibliothèque standard de Coq. Le tableau de la figure 1 répertorie les

différentes fonctions de notre développement selon leur origine ou le besoin. La dernière colonne indique dans quel fichier la fonction a été définie. Ces fichiers sont accessibles directement en cliquant sur leur nom.

Fonction	FaCiLe	CoqBinFD	FSet	Définition dans
create	X	X		*create.v*
values	X		X	*structure.v*
size		X	X	*structure.v*
is_empty	X	X	X	*structure.v*
is_singleton		X		*operation_solver_basic.v*
choose	X	X	X	*operation_solver_basic.v*
fold_left		X	X	*operation_solver_fold.v*
exist		X	X	*operation_solver_exist_forall.v*
for_all		X	X	*operation_solver_exist_forall.v*
included	X	X	X	*operation_solver_included.v*
member	X	X	X	*operation_solver_member.v*
remove	X	X	X	*operation_solver_remove.v*
equal			X	*equal.v*
partition	X		X	*operation_solver_filter_partition.v*
filter		X	X	*operation_solver_filter_partition.v*
add	X		X	*operation_FSet_union.v*
union	X		X	*operation_FSet_union.v*
intersection	X		X	*operation_FSet_intersection.v*
difference	X		X	*operation_FSet_difference.v*

Figure 1: Liste des fonctions définies et prouvées correctes en Coq

4.1 Structure des domaines

La représentation utilisée d'un domaine est constituée essentiellement d'une liste d'intervalles, de type `elt_list`, stockée dans un enregistrement (champ `domain`). Pour faciliter certaines opérations liées à la résolution des contraintes, la valeur minimale et la valeur maximale sont stockées dans les champs respectifs `min` et `max`, ainsi que la taille, dans le champ `size`. Le terme *taille* ici, correspond au nombre de valeurs présentes dans le domaine. Par exemple, la taille de la liste d'intervalles `[(1, 10)]` est 10. Les types `elt_list` et `t` sont définis dans le listing 1.

```
1 Inductive elt_list :=
2 | Nil : elt_list
3 | Cons : Z -> Z -> elt_list -> elt_list.
4
5 Record t := mk_t {  domain : elt_list ; size : Z ; max : Z ; min : Z }.
```

Listing 1: Types d'une liste d'intervalles et d'un domaine

Cependant, le champ `domain` doit contenir une liste d'intervalles avec la particularité que les intervalles doivent être rangés dans l'ordre croissant, tous disjoints et non vides. De plus, cette liste doit avoir un nombre d'intervalles minimum. Cette condition de bonne formation, appelée `invariant` dans la suite, est formalisée à l'aide du prédicat `Inv_elt_list` dont la définition est donnée dans le listing 2. Le prédicat (`Inv_elt_list j l`) spécifie plus généralement les

conditions précédentes sur la liste d'intervalles `l` et, que les entiers contenus sont tous supérieurs ou égaux à la borne `j`.

```
Inductive Inv_elt_list : Z -> elt_list -> Prop :=
| invNil : forall j, Inv_elt_list j Nil
| invCons : forall (a b j : Z) (q : elt_list),
      j <= a -> a <= b ->  Inv_elt_list (b+2) q ->
      Inv_elt_list j (Cons a b q).
```

Listing 2: Invariant pour une liste d'intervalles

L'invariant ou la bonne formation du domaine, `Inv_t` dont le code Coq est donné ci-dessous (listing 3), s'en déduit alors simplement : le champ `domain` doit être bien formé au sens de l'invariant précédemment défini, le champ `min` (resp. `max`) doit contenir le plus petit (resp. grand) élément de la liste d'intervalles définie au champ `domain`, enfin la valeur du champ `size` doit correspondre à la taille de la liste d'intervalles définie au champ `domain`. Les fonctions `get_min`, `process_max` et `process_size` permettent d'obtenir respectivement le minimum, le maximum et la taille d'une liste d'intervalles donnée. La valeur `min_int` dans la définition de `Inv_t` est une valeur par déefaut, paramètre non contraint de la formalisation. FaCiLe utilise la borne inférieure des entiers et c'est pour cette raison que notre constante s'appelle ainsi.

```
Definition Inv_t (d : t) := Inv_elt_list (min d) (domain d)
                            /\ (min d) = get_min (domain d) min_int
                            /\ (max d) = process_max (domain d)
                            /\ (size d) = process_size (domain d).
```

Listing 3: Invariant pour la structure de domaine

Par la suite, nous vérifierons que les fonctions définies établissent ou préservent ces invariants.

4.2 Vérification formelle des fonctions

De manière générale, pour la plupart des fonctions sur les domaines, est définie une fonction analogue sur une liste d'intervalles, comme dans FaCiLe. Par exemple, la fonction `remove` qui permet de retirer un élément d'un domaine (opération effectuée lors du filtrage par exemple), de type `Z -> t -> t`, est définie en utilisant la fonction `elt_list_remove` qui supprime un élément éventuellement présent dans une liste d'intervalles et retourne un booléen indiquant si la suppression a été effective ou non. Enlever un élément d'une liste d'intervalles consiste à rechercher l'intervalle $\mathcal{I}$ susceptible de contenir l'élément à supprimer, et selon les cas, à supprimer l'intervalle (cas où $\mathcal{I}$ est un singleton), modifier une des bornes (cas où l'élément était une des bornes de $\mathcal{I}$), ou scinder $\mathcal{I}$ en deux intervalles. La fonction `remove` s'occupe aussi de calculer les valeurs minimale et maximale, ainsi que la longueur. De même, les fonctions `elt_list_member` et `member` testent l'appartenance d'un élément à, respectivement, une liste d'intervalles (voir listing 4) ou un domaine. La première explore la liste jusqu'à trouver un intervalle contenant l'élément recherché ou une borne strictement supérieure à ce dernier.

```
Fixpoint elt_list_member (x : Z) (l : elt_list) : bool := match l with
|Nil => false
|Cons min max q => if x <=? max then  x >=? min
                   else elt_list_member x q
end.
```

Listing 4: Appartenance d'un élément à une liste d'intervalles

Du point de vue vérification, chaque fonction est accompagnée d'une preuve de préservation ou établissement de l'invariant associé (`Inv_elt_list` ou `Inv_t` selon que la fonction travaille sur une liste d'intervalles ou un domaine). Elle est également accompagnée d'un théorème de correction dont l'énoncé est inspiré de celui de l'interface des ensembles finis de Coq (`Fsetinterface`). Les différents théorèmes pour les fonctions `elt_list_remove` et `remove` sont donnés, pour illustration, dans le listing 5.

```
1  Lemma elt_list_remove_inv : forall (l1 : elt_list) (e : Z) l2 b y,
2  elt_list_remove l1 e = (b, l2) ->
3  Inv_elt_list y l1 ->
4  Inv_elt_list y l2.
5
6  Theorem remove_inv : forall (d : t),
7  Inv_t d -> forall (e : Z), Inv_t (remove e d).
8
9  Theorem elt_list_remove_spec_1 : forall l x y,
10 Inv_elt_list y l ->
11 elt_list_member x (snd (elt_list_remove l x)) = false.
12
13 Theorem remove_spec_1 : forall d v,
14 Inv_t d -> member v (remove v d) = false.
15
16 Theorem elt_list_remove_spec_2_3 : forall l v w y,
17 Inv_elt_list y l -> w <> v ->
18 elt_list_member w (snd (elt_list_remove l v)) = elt_list_member w l.
19
20 Theorem remove_spec_2_3 : forall d v y,
21 Inv_t d -> y <> v ->
22 member y (remove v d) = member y d.
```

Listing 5: Théorèmes relatifs aux fonctions `elt_list_remove` et `remove`

4.3 Zoom sur la fonction `create`

La fonction `create` (voir son code dans le listing 6) prend en paramètre une liste d'entiers l et crée le domaine correspondant. La liste l est quelconque, elle peut donc contenir des doublons et ne pas être triée dans l'ordre croissant. La fonction `create` agit de la façon suivante :

- elle trie la liste l dans l'ordre croissant en éliminant les doublons, en appliquant la fonction `sortWithoutDup` qui est ici une adaptation du tri par insertion (alors que le code OCaml de FaCiLe utilise la fonction `List.sort` de la bibliothèque standard - plus efficace que notre fonction de tri - et ensuite enlève les doublons),
- elle crée ensuite le domaine par un appel à la fonction `unsafe_create`.

```
1  Definition create (Zl : list Z) : t := unsafe_create (sortWithoutDup Zl).
```

Listing 6: Définition de la fonction `create`

L'objectif, à présent, est de montrer que cette fonction établit bien l'invariant précédemment défini, c'est-à-dire que le domaine créé respecte l'invariant.

Un premier travail consiste alors à montrer que la fonction `sortWithoutDup` est correcte, c'est-à-dire qu'elle produit une liste triée dans l'ordre croissant sans doublons, et que la liste produite contient les mêmes éléments que la liste initiale. Pour ce faire, nous avons utilisé le prédicat `NoDup` de la bibliothèque standard de Coq, adapté le prédicat `Sorted` à nos besoins, et introduit un prédicat `SameElt` formulant que 2 listes ont les mêmes éléments :

```
Definition SameElt l1 l2 : Prop := forall (x : Z), In x l1 <-> In x l2.
```

Listing 7: Prédicat formulant que 2 listes ont les mêmes éléments ou non

Nous pouvons alors écrire la spécificationde la fonction `sortWithoutDup` :

```
Theorem sortWithoutDup_spec :
forall l, Sorted (sortWithoutDup l)
        /\ SameElt l (sortWithoutDup l)
        /\ NoDup (sortWithoutDup l).
```

Listing 8: Spécification de la fonction `sortWithoutDup`

Ensuite, nous avons montré que la fonction `unsafe_create` est correcte, c'est-à-dire qu'à partir d'une liste triée dans l'ordre croissant et sans doublons, elle construit un domaine respectant l'invariant précédemment établi. Cette fonction utilise la fonction `make` pour créer la liste d'intervalles correspondant à une liste d'entiers supposée triée dans l'ordre croissant et sans doublons. En effet, la fonction `make a b l` parcourt la liste l à la recherche d'une suite $b+1, b+2 \ldots b+n$ jusqu'à une rupture, et construit alors l'intervalle de a à $b+n$ et recommence avec les éléments suivants de l.
Pour démontrer que `create` établit l'invariant concernant la liste d'intervalles, nous avons prouvé le théorème suivant qui porte sur la fonction `make` :

```
Theorem elt_list_make_inv : forall l a b j,
Sorted l -> NoDup l -> j <= a -> a <= b ->
(forall x, In x l -> b+1 <= x) -> Inv_elt_list j (make a b l).
```

Listing 9: Théorème essentiel sur la fonction `make`

Nous avons donc ainsi pu démontrer que la fonction `create` produit un domaine qui respecte l'invariant :

```
Theorem create_inv : forall l, Inv_t (create l).
```

Listing 10: Établissement de l'invariant par la fonction `create`

Il reste à démontrer la correction de la fonction `create` (listing 11), à savoir l'équivalence entre l'appartenance au domaine `create l` et l'appartenance à la liste initiale l (`In` est ici le prédicat d'appartenance à une liste de la bibliothèque standard de Coq).

```
Theorem create_spec : forall l,
(forall y, member y  (create l) = true <-> In y l).
```

Listing 11: Spécification de la fonction `create`

Ce théorème découle de propriétés sur la fonction `make`, telle que celle présentée dans le listing 12, qui énonce que tout élément de la liste d'intervalles `make a b l` est, soit un entier

compris entre `a` et `b`, soit un élément de `l`. On démontre également la propriété réciproque avec des hypothèses supplémentaires spécifiant que la liste `l` ne doit contenir que des entiers strictement supérieurs à `b` et que cette liste est triée et sans doublons.

```
1 Lemma elt_list_member_make : forall l a b, a <= b ->
2 forall y, elt_list_member y (make a b l) = true ->
3 a <= y <= b \/ In y l.
```

Listing 12: Propriété essentielle sur la fonction `make` pour prouver la spécification de `create`

Dans cette partie du développement formel, la difficulté a été de retrouver les hypothèses nécessaires pour réussir à démontrer les propriétés.

4.4 Opérations ensemblistes classiques

Les opérations ensemblistes telles que l'intersection, l'union et la différence, présentes dans FaCiLe ont été traduites et prouvées correctes par rapport aux spécifications *classiques* de la théorie des ensembles énoncées dans l'interface `FSetinterface`. Toutes ces fonctions, lorsqu'elles sont définies au niveau des listes d'intervalles, utilisent une récursion générale s'appuyant sur la décroissance de la taille d'une des deux listes en entrée. Les preuves de terminaison sont très simples. Les preuves de préservation d'invariant et de correction sont fastidieuses et longues, requérant de s'intéresser à de nombreux cas. Il faut noter que, lors de la traduction, la fonction qui calcule l'intersection des domaines `s1` et `s2` a été simplifiée car elle commence par quelques optimisations (si les deux ensembles sont égaux, alors l'intersection est l'un des ensembles; si la taille de la liste d'intervalles de l'intersection est égale à la taille de `s1` (resp. `s2`), alors l'intersection est égale à `s1` (resp. `s2`)) qui requièrent des démonstrations que nous n'avons pas eu le temps de faire. Ces dernières entrent dans les perspectives de ce travail. Une simplification similaire (comparaison préalable des tailles des domaines et des valeurs minimales et maximales) a été faite, pour le code de la fonction `included` qui teste si un domaine est inclus dans un autre, par manque de temps également. Une perspective est bien sûr de considérer ces optimisations, néanmoins le code Coq devra distinguer les cas des ensembles vides, ce que ne fait pas le code FaCiLe, car il s'appuie sur le fait que `min_int` est inférieur ou égal à tout `int`. La preuve des propriétés de correction de la fonction `elt_list_included` a permis de proposer une version plus efficace de cette fonction, version utilisée dans les expérimentations présentées dans la section 5.

4.5 For_all, Exist, Partition, Filter, Fold

Ces fonctions ne sont pas présentes dans le module `Domain` de FaCiLe. Elles sont néanmoins nécessaires dans le solveur CoqBinFD, par exemple pour réaliser un filtrage, et sont requises par l'interface `FSetinterface` de Coq. Elles suivent un schéma légèrement différent de celui des fonctions précédentes, car il est besoin d'explorer chaque valeur des intervalles. On aurait pu, en utilisant la fonction `values` obtenir la liste des éléments et ensuite appliquer les fonctions correspondantes sur les listes. Néanmoins ce processus algorithmique se révèle trop coûteux.

La fonction `filter` doit construire un domaine à partir d'un domaine argument et d'une fonction booléenne. La fonction `elt_list_filter`, qui travaille sur les listes d'intervalles, construit, pour chaque intervalle présent dans la liste de départ, un ou plusieurs intervalles qu'elle assemble ensuite. Par exemple, si la liste d'intervalles est `[(1, 6);(40, 43)]` la fonction qui filtre tous les entiers pairs construira la liste `[(2, 2);(4, 4);(6, 6);(40, 40);(42, 42)]`. La fonction est inspirée de la fonction `make` précédemment décrite. La fonction `partition p`

`d` fait deux appels à `filter`, l'un pour construire le domaine qui contient les éléments de `d` qui satisfont le prédicat `p`, l'autre pour construire le domaine des éléments de `d` qui ne satisfont pas `p`. Cette implantation n'est certes pas efficace, l'optimisation qui consiste à construire les deux domaines simultanément reste à faire mais devrait suivre un schéma similaire.

4.6 Conformité à l'interface `FSet` de Coq

Une nouvelle implantation des ensembles finis d'entiers, qui s'appuie sur les domaines utilisant les listes d'intervalles, est disponible (dans le fichier `bridge.v`). Le module définit le type `t_FSet` (défini dans le listing 13, où le type `structure.t` désigne le type des domaines défini dans le listing 1) comme un enregistrement contenant un domaine et une preuve de sa bonne formation. Chaque fonction de ce module encapsule le résultat de la fonction correspondante sur les domaines, ainsi que la preuve de préservation ou d'établissement de l'invariant le cas échéant.

```
1 Record t_Fset := mk_t_Fset {
2   set : structure.t ;
3   wf : Inv_t set
4 }.
```

Listing 13: Type d'un domaine dans notre module

Le code de `remove` est donné en illustration dans le listing 14.

```
1 Definition remove x s :=
2   let res := operation_solver.remove x (set s) in
3   mk_t_Fset res (remove_inv (set s) (wf s) x).
```

Listing 14: Définition de la fonction `remove` dans notre module

La relation d'ordre permettant d'ordonner totalement les ensembles est définie à partir d'une relation d'ordre totale portant sur les listes ordonnées d'entiers via la fonction `values` (`elements` selon l'interface `FSetinterface`).

5 Extractions et expérimentations

Le mécanisme d'extraction de Coq permet d'obtenir un code OCaml exécutable, mécanisme que nous avons utilisé en vue de pouvoir utiliser les listes d'intervalles comme représentation des domaines avec le solveur de contraintes CoqBinFD.

Nous présentons dans cette section quelques tests que nous avons réalisés avec le code extrait afin de comparer les temps d'exécution obtenus, en fonction notamment de la forme du domaine. Nous comparons également ces temps d'exécution avec les temps obtenus en réalisant les mêmes tests avec la bibliothèque FaCiLe.

Le mécanisme d'extraction de Coq permet de faire quelques ajustements dans le code extrait, par exemple remplacer certains types (ou fonctions) définis en Coq par des types (ou fonctions) de OCaml, et par conséquence, dans certains cas, *casser* la correction du code extrait. La formalisation en Coq présentée dans cet article utilise les entiers mathématiques (de type `Z`) alors que la bibliothèque FaCiLe utilise les `int` de OCaml, entiers bornés *modulo*. L'extraction des entiers mathématiques vers les entiers de OCaml, dans laquelle la définition du type Z (resp. certaines fonctions de calcul sur `Z`) est remplacée par le type `int` (resp. par

des fonctions OCaml manipulant des `int`), n'est en ce sens pas correcte : le code extrait peut, par exemple, occasionner des dépassements de capacité alors que le calcul en Coq ne pose pas de problème. Néanmoins nous utilisons cette extraction incorrecte[2] pour pouvoir comparer, en termes de temps d'exécution, le code issu de notre développement avec le code FaCiLe. Dans la suite, ce code extrait est appelé code CoqInt. Nous présentons également une extraction dite propre (ou extraction vers Z), où la traduction de la définition de Z est utilisée dans le code extrait - appelé dans la suite CoqZ - ainsi qu'une extraction vers les grands entiers de OCaml[3]. Le code extrait issu de cette dernière extraction est appelé CoqBG. Notons que, dans chaque expérience d'extraction, nous avons remplacé les booléens de Coq par les `bool` de OCaml et leurs opérations `andb` et `orb` par les opérateurs paresseux de OCaml correspondants.

Pour nos tests de comparaison, quatre catégories de domaines ont été définies, chacune contenant 5 domaines :

- *Petits domaines* : moins de 10 valeurs, c'est-à-dire `size` ≤ 10;
- *Grands domaines* : plus de 100 valeurs, c'est-à-dire `size` ≥ 100;
- *Domaines compacts* : peu de *trous* (`size` $= 100$).
- *Domaines creux* : des valeurs éparpillées (`size` $= 100$);

Cela permet d'avoir de nombreux cas de figure des domaines possibles, ce qui aurait été plus compliqué à maîtriser si les domaines avaient été générés aléatoirement.

Les résultats obtenus sont présentés, par catégorie, sur les graphiques des figures 3, 4, 5 et 6. Les fonctions énumérées dans le tableau de la figure 1 sont exécutées sur chacun des domaines. Chaque valeur correspond au temps d'exécution moyen de 1 000 itérations par domaine. L'unité utilisée est la seconde. Dans les graphiques 4, 5 et 6, la fonction `create` n'apparaît pas pour des raisons d'échelle, tous ses temps d'exécution sont reportées sur le graphique de la figure 2.

Les tests ont été réalisés sur un MacBook Pro *2.3 GHz Intel Core I5* avec *8 GB 2133 MHz LPDDR3*, avec OCaml 4.05 et Coq 8.9.1.

5.1 Comparaison CoqInt - FaCiLe

La comparaison ne porte ici que sur les fonctions communes aux deux codes. Lorsque nous utilisons des *petits domaines*, les temps obtenus en exécutant le code FaCiLe et le code extrait CoqInt sont sensiblement les mêmes, à l'avantage de FaCiLe sauf pour dans le cas de la fonction `is_empty` qui, pourtant, ne réalise qu'un accès à un champ d'un enregistrement dans les deux codes. Concernant les *grands domaines*, les temps obtenus sont sensiblement les mêmes, sauf pour `create`. En effet, la fonction `create` de FaCiLe est 25 fois plus rapide que celle du code Coq extrait. Nous payons ici le choix d'un tri moins efficace. Avec des *domaines compacts*, nous constatons les mêmes résultats. Enfin, pour les *domaines creux*, les temps obtenus restent également proches. Les mêmes tests ont été réalisés sur la fonction `create` en modifiant dans les codes extraits de Coq le tri vérifié formellement par un appel à `List.sort` et la fonction de FaCiLe qui enlève les doublons. Les résultats sont montrés à la figure 2, pour les catégories des domaines grands, compacts et creux. Les temps d'exécution pour les petits domaines n'apparaissent pas car ils sont très inférieurs aux autres. La barre int (resp. Z et big_int) concrétise le temps pour le code extrait CoqInt (resp. CoqZ et CoqBG) alors que la barre

[2] Nous avons utilisé le module `Coq.extraction.ExtrOcamlZInt` de la bibliothèque standard de Coq
[3] en utilisant le module `Coq.extraction.ExtrOcamlZBigInt` de la bibliothèque standard de Coq

Z/sort désigne le même code avec la fonction non certifiée de tri. Il apparaît clairement que le choix du changement de tri rend le code moins efficace lors de la création de domaines de grande taille.

Nous avons également réalisé quelques tests aux limites, par exemple la création du domaine le plus grand possible, à l'aide de la fonction `interval`, soit `interval min_int max_int`. Le calcul de la taille de ce domaine est, sans surprise, égale à 0 en exécutant le code FaCiLe et le code CoqInt. Un problème est également détecté lors du calcul de l'union du domaine de valeurs 0 et 1 (`boolean`) et du domaine réduit à l'intervalle `min_int` .. `min_int` + 2 (`interval min_int (min_int + 2)`). Les intervalles du résultat sont les bons mais mal ordonnés dans la liste, alors que l'union de `(interval min_int (min_int + 2))` et de `boolean` est calculée correctement. Des problèmes de dépassement arithmétique sont également constatés sur la différence, dès lors qu'une borne `min_int` ou `max_int` est utilisée. Ces tests s'exécutent, bien entendu, sans problème dans Coq et avec les autres extractions, puisque `min_int` et `max_int` y sont des entiers comme les autres. Cette possibilité de dépassement de capacité arithmétique est mentionnée dans la documentation de FaCiLe sous la forme de la petite note suivante : *Users should be careful when expecting the arithmetic solver to compute bounds from variables with very large domain, that means with values close to max_int or min_int (depending on the system and architecture).*

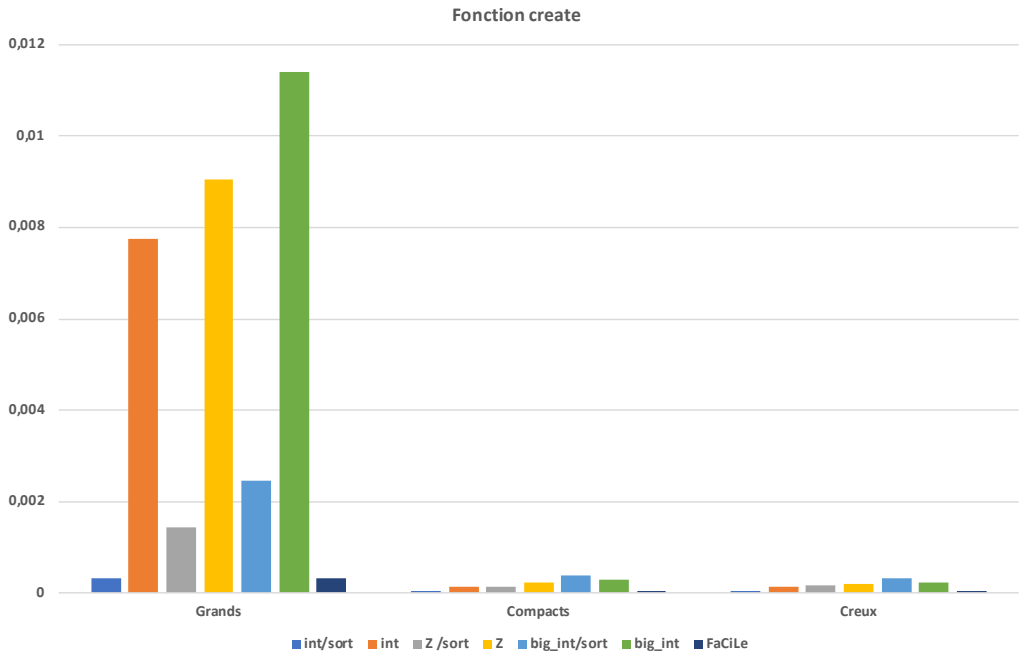


Figure 2: Temps d'exécution moyen en seconde pour 1 000 itérations de la fonction `create` (axe horizontal) sur 5 domaines des catégories Grands, Compacts et Creux

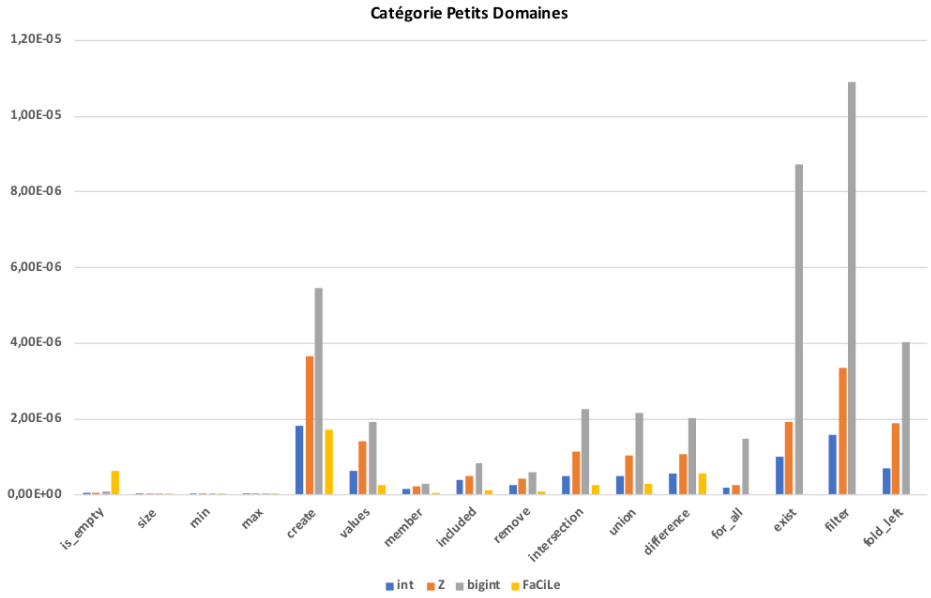


Figure 3: Temps d'exécution moyen en seconde pour 1 000 itérations des fonctions sur 5 domaines de la catégorie *Petits domaines* pour les 3 codes extraits et le code FaCiLe

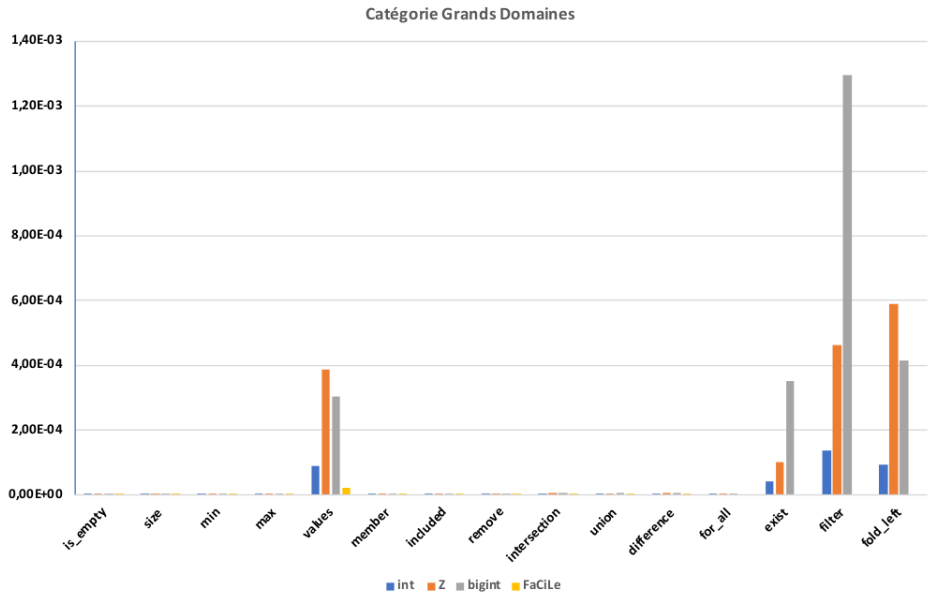


Figure 4: Temps d'exécution moyen en seconde pour 1 000 itérations des fonctions sur 5 domaines de la catégorie *Grands domaines* pour les 3 codes extraits et le code FaCiLe

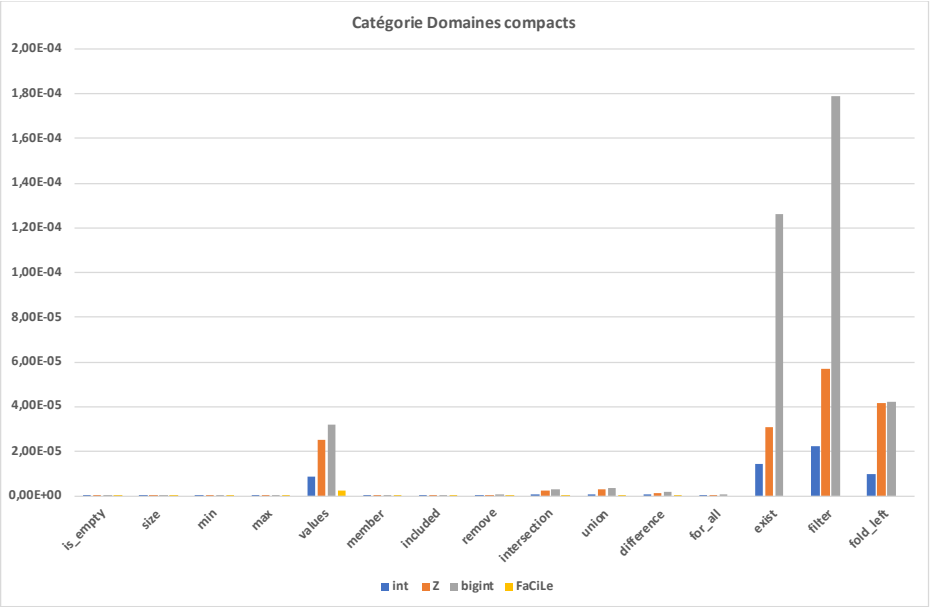


Figure 5: Temps d'exécution moyen en seconde pour 1 000 itérations des fonctions sur 5 domaines de la catégorie *Domaines compacts* pour les 3 codes extraits et le code FaCiLe

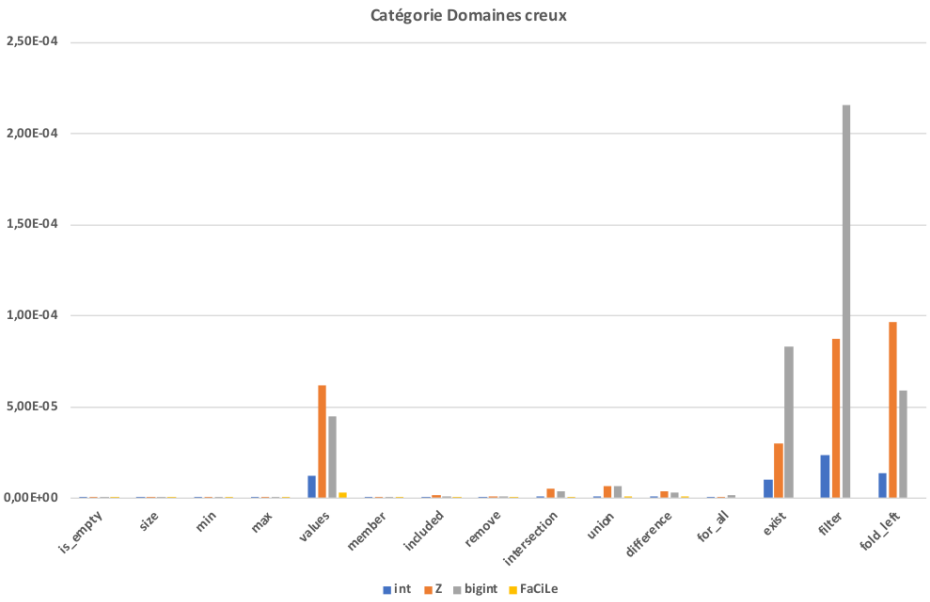


Figure 6: Temps d'exécution moyen en seconde pour 1 000 itérations (axe horizontal) sur 5 domaines de la catégorie *Domaines creux* pour les 3 codes extraits et le code FaCiLe

5.2 Comparaison CoqInt - CoqZ et CoqBG

Pour toutes les catégories, on peut constater, de manière générale, que le temps d'exécution pour le code extrait CoqBG est plus important que celui obtenu avec CoqZ, lui-même supérieur à celui obtenu avec CoqInt. Pour la plupart des fonctions, les temps de calcul obtenus avec l'extraction propre, CoqZ, restent raisonnables. Les fonctions qui parcourent toutes les valeurs du domaine, par exemple `filter`, ont un temps d'exécution trop important sur les domaines grands et compacts. Ce temps dépend aussi des calculs faits sur chaque entier du domaine.

Nous pouvons distinguer 4 groupes de fonctions et donc 4 profils d'exécution. Les fonctions `is_empty`, `size`, `min` et `max` accèdent à un champ de la structure et sont donc réalisées en temps constant. Les fonctions ensemblistes, `member`, `included`, `remove`, `union`, `intersection` et `difference`, parcourt la liste des intervalles et font quelques comparaisons relativement aux bornes des intervalles. Le temps d'exécution est donc linéaire par rapport au nombre d'intervalles. Le troisième groupe regroupe les fonctions `for_all`, `exist`, `filter`, `fold_left` et `values` qui requièrent d'énumérer, une à une, les valeurs du domaine, en totalité ou non selon les fonctions. Les temps d'exécution sont par conséquent liés à la taille de l'ensemble fini encodé. Enfin le quatrième groupe est composé de la fonction `create` déjà évoquée plus haut, qui requiert l'utilisation d'un algorithme de tri efficace.

6 Conclusion

Le développement formel que nous avons présenté dans cet article permet de fournir une implantation vérifiée formellement des domaines d'entiers contenant les fonctions nécessaires à une nouvelle implantation du solveur CoqBinFD. Ce travail fournit également une implantation formellement vérifiée des ensembles finis d'entiers utilisable dans tout développement Coq. Il est également un début de vérification formelle du module dédié à la représentation des domaines de la bibliothèque OCaml existante FaCiLe.

Au total, le développement formel Coq compte environ 7 750 lignes de code (hors le module faisant le lien avec `FSetinterface`) et comprend 2 définitions de type, 2 prédicats logiques qui définissent la bonne formation des listes d'intervalles et des domaines, 70 définitions de fonctions dont 57 sont issues de la traduction en Coq de fonctions OCaml extraites de FaCiLe. Environ 263 lemmes et théorèmes ont été démontrés. Le fichier `bridge.v` ajoute environ 950 lignes de code, la majeure partie de ses définitions et lemmes étant des *redirections* vers des définitions et lemmes existants. Le code Coq ainsi que les fichiers de test sont disponibles à l'adresse suivante `https://gitlab.com/finite_set_Coq/real_intervals_list`.

Nous envisageons de poursuivre ce travail en prenant en compte les quelques optimisations *oubliées* par manque de temps et en reliant cette représentation des domaines au solveur CoqBinFD. De manière générale nous en envisageons une refonte plus modulaire, permettant ainsi d'être indépendant de la représentation des domaines utilisée. La variante utilisant la consistance de bornes décrite dans [11] pourrait également bénéficier utilement du développement formel présenté ici, car un accès efficace aux bornes y est crucial. Une autre perspective importante concerne l'obtention d'un code OCaml extrait efficace. Nous projetons pour cela de reprendre le développement formel en utilisant les entiers cycliques de la bibliothèque Coq, ce qui nous permettrait d'avoir un code extrait efficace et certifié. Ce travail contribue à la vérification formelle de bibliothèques OCaml existantes, pour cette raison nous avons tenu à rester le plus proche possible de l'implantation existante. Néanmoins une perspective de ce travail est de généraliser la notion de domaine à un autre support que celui des entiers, per-

mettant ainsi de définir et manipuler des domaines sur un type fini quelconque. Enfin, un travail envisagé à plus long terme est de formaliser les autres représentations utilisées dans les solveurs existants (cf section 2), en particulier les représentations qui se concentrent sur les valeurs absentes (*gap interval list* ou *gap interval tree*) avec un objectif de réutilisation des preuves effectuées dans le cadre du développement formel présenté dans l'article.

Remerciements : nous remercions vivement les rapporteurs anonymes qui, par leurs commentaires, nous ont permis d'améliorer cet article. Ce travail a été financé partiellement par la direction de la recherche de l'ENSIIE.

Bibliographie

[1] Documentation du solveur ilog solver. http://lia.deis.unibo.it/Courses/AI/applicationsAI2009-2010/materiale/cp15doc/usrsolver/usrsolverpreface.html.

[2] Github du solveur naxos solver. https://github.com/pothitos/naxos.

[3] Site officiel de minizinc. https://www.minizinc.org/.

[4] Site officiel du solveur eclipse prolog. http://eclipseclp.org/.

[5] Site officiel du solveur gecode. http://www.gecode.org/.

[6] Site officiel du solveur minion. https://constraintmodelling.org/minion/.

[7] Site officiel du solveur sicstus prolog. https://sicstus.sics.se/.

[8] P. Brisset and N. Barnier. FaCiLe : a Functional Constraint Library. In *CICLOPS 2001, Colloquium on Implementation of Constraint and LOgic Programming Systems*, Paphos, Cyprus, 2001.

[9] M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 116–131, Paris, France, 2012.

[10] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011.

[11] C. Dubois and A. Gotlieb. Solveurs cp(fd) vérifiés formellement. In *Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 115–118, 2013.

[12] C. Guillaume. Coq of OCaml. The OCaml Users and Developers Workshop, Gothenburg, Sweden, September 5, 2014.

[13] V. Le clément de saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[14] N. Pothitos and P. Stamatopoulos. Flexible management of large-scale integer domains in csps. In *Artificial Intelligence: Theories, Models and Applications, 6th Hellenic Conference on AI, SETN 2010, Athens, Greece*, volume 6040 of *LNCS*, pages 405–410. Springer, 2010.

[15] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. http://www.choco-solver.org.

[16] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 495–526. Elsevier, 2006.

Nombres réels dans Coq

Vincent Séméria
email (lambda x y ⇒ x.y@gmail.com) vincent semeria

Paris

Abstract

Les nombres réels de la bibliothèque standard étaient axiomatisés, de façon semi-classique, jusqu'à la version 8.10 de Coq. Ceci posait 3 problèmes : la cohérence de ces axiomes n'était pas formellement prouvée (elle était informellement dérivable du modèle ensembliste de Coq [13][17]), ces axiomes bloquaient les calculs (la réduction sous forme normale des approximations rationnelles) et enfin l'utilisateur ne pouvait pas se restreindre aux nombres réels constructifs.

Nous présentons une découpe des nombres réels de Coq en une première couche constructive et sans axiome, suivie d'une seconde couche qui donne les nombres réels classiques par quotient de la première couche. L'absence d'axiome dans la couche constructive établit sa cohérence avec le noyau de Coq, c'est-à-dire le calcul des constructions inductives [11]. La seconde couche classique introduit 3 axiomes logiques, qui sont usuels et vrais dans le modèle ensembliste de Coq, leur cohérence est donc prouvée. Enfin l'utilisateur a désormais le choix d'utiliser les nombres réels constructifs ou classiques, selon quels fichiers de la bibliothèque standard il importe.

Cette découpe a été intégrée dans la bibliothèque standard de Coq le 27 octobre 2019, pour une sortie dans la prochaine version 8.11. Elle est rétro-compatible : toutes les bibliothèques (Coquelicot, Flocq, VST, ...) qui utilisaient les nombres réels de Coq jusqu'à la version 8.10 continuent de compiler et de fonctionner, sans aucune modification de leur code source.

1 Introduction

Les nombres réels servent à mesurer l'espace qui nous entoure. Ils complètent les nombres rationnels : des longueurs très simples telles que la circonférence du cercle sont des nombres réels, irrationnels et même transcendants (non racines de polynômes à coefficients entiers). Depuis Archimède, on sait que π s'obtient comme limite de polygones, c'est-à-dire une suite $u : \mathbb{N} \to \mathbb{Q}$ dont les valeurs convergent. La convergence peut se formaliser par le critère de Cauchy,

$$\forall n \in \mathbb{N}^*,\ \exists k \in \mathbb{N},\ \forall i, j \geq k,\ |u_i - u_j| \leq \frac{1}{n} \tag{1}$$

Des variantes sont possibles : il existe une fonction $f : \mathbb{N}^* \to \mathbb{N}$ telle que $\forall n \in \mathbb{N}^*, \forall i, j \geq f(n), |u_i - u_j| \leq \frac{1}{n}$. Et nous pouvons de surcroît demander que la fonction f soit calculable par une machine de Turing. Nous obtenons des qualités croissantes de nombres réels, i.e. des inclusions décroissantes.

Le cas calculable, qui est le meilleur, donne un algorithme déterministe pour approximer la limite par un nombre rationnel, à la précision que l'on veut. Le pire cas (1) laisse l'entier k entre les mains d'un "oracle", on n'a aucun moyen d'en connaître la valeur. C'est le cas de certaines suites croissantes et majorées, par exemple les suites de Specker [16], pour lesquelles f est non calculable.

Dans Coq, π a été calculé de différentes façons, par exemple [3]. Boldo, Lelay et Melquiond [6] ont d'autre part recensé l'état de l'art de l'analyse réelle, dans plusieurs assistants de preuve

dont Coq. Pour les nombres réels de la bibliothèque standard de Coq, nous souhaitons donner le choix de se restreindre aux nombres calculables, ou au contraire de considérer des nombres plus exotiques, éventuellement non calculables. Il suffit pour cela de jouer sur la logique de Coq. Si nous ne posons aucun axiome et nous contentons du noyau de Coq, alors nous bénéficions de la propriété de canonicité : le calcul d'un terme jusqu'à sa forme normale aboutit à un constructeur de ce terme. Nous pensons que la canonicité est une condition nécessaire pour les mathématiques constructives[1], qui réclament un témoin "explicite" dans chaque preuve d'existence. À titre d'exemple, la preuve de l'existence de 2 nombres réels irrationnels a, b tels que la puissance a^b soit rationnelle. Une première preuve non constructive pourrait supposer par l'absurde la non existence de a, b, puis en dériver une contradiction. Dans ce cas on n'aurait aucune idée des valeurs de a et b. Une preuve un peu plus précise invoque l'axiome du tiers exclu, en affirmant que ou bien $\sqrt{2}^{\sqrt{2}}$ est rationnel, ou bien il ne l'est pas. S'il l'est alors nous avons trouvé $a = b = \sqrt{2}$. Sinon, nous prenons $a = \sqrt{2}^{\sqrt{2}}$ et $b = \sqrt{2}$. Nous avons ici beaucoup plus d'informations, puisque nous savons que b vaut $\sqrt{2}$, et que a vaut ou bien $\sqrt{2}$, ou bien $\sqrt{2}^{\sqrt{2}}$. Mais nous voyons qu'il reste du travail pour a, dont le calcul est bloqué par l'axiome du tiers exclu : on n'a pas la forme canonique de a, les preuves utilisant le tiers exclu ne sont pas constructives. Conformément à l'usage, nous appelons "classiques" les nombres réels où l'usage d'axiomes logiques tels que tiers exclu est autorisé.

Nous pourrions également prouver des théorèmes sur les réels classiques, puis étudier a posteriori si un certain réel classique x est calculable. Néanmoins, des mathématiciens tels que Bishop [4] pourraient insister pour la preuve qu'un algorithme A calcule un nombre réel x soit elle aussi constructive. Par exemple la preuve que le nombre 7 vérifie une certaine propriété P peut se faire de façon classique par analyse-synthèse. C'est-à-dire qu'on commence par prendre n'importe quel nombre x vérifiant P, puis en raisonnant on trouve que x doit être égal à 7, ce qui conclut l'analyse. Pour la synthèse, on montre éventuellement par l'absurde qu'il existe un nombre x vérifiant P. On déduit de ces deux parties que le nombre 7 vérifie P. Nous pouvons objecter qu'une telle preuve est moins convaincante qu'une "preuve directe" que 7 vérifie P. Le refus constructif du tiers exclu dans la partie synthèse est un des moyens de clarifier ce que nous entendons par "preuve directe".

Plan. La section 2 présente les deux couches de nombres réels. La section 3 illustre comment une preuve constructive passe semi-automatiquement dans la couche classique. Enfin, la section 4 discute les calculs et leur performance.

2 Nombres réels

2.1 Constructifs

Pour les nombres réels constructifs, nous nous sommes inspirés de la bibliothèque de mathématiques constructives C-CoRN [1][10][15][12]. Ses nombres réels sont des suites de Cauchy de nombres rationnels, dont nous avons déjà parlé en (1), et qui sont également expliquées par Bishop [4]. Cette suite récurrente est de Cauchy dans les nombres rationnels

$$x_0 = 2, \quad x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n}$$

[1]Nous n'affirmons nullement que le calcul des constructions inductives est le seul cadre pour faire des mathématiques constructives. On peut se reporter aux variantes axiomatiques entre Brouwer, Bishop et Markov par exemple, ainsi qu'aux diverses variantes des théories des types.

et elle converge vers $\sqrt{2}$. Ainsi les suites de Cauchy servent à construire depuis $\mathbb{Q}$ les limites qui manquent à $\mathbb{Q}$ (ici $\sqrt{2}$). Illustrons simplement la proposition que deux suites de nombres rationnels sont de Cauchy et ont la même limite (donc définissent le même nombre réel), extraite du fichier `Coq.Reals.ConstructiveCauchyReals`,

```
Definition QSeqEquiv (un vn : nat -> Q) (cvmod : positive -> nat) : Prop
  := forall (k : positive) (p q : nat),
      cvmod k <= p -> cvmod k <= q
      -> Qabs (un p - vn q) < (1 # k).
```

`positive` est le type des entiers strictement positifs. `Qabs` est la valeur absolue sur $\mathbb{Q}$. `1#k` est la fraction $\frac{1}{k}$. `cvmod` est la fonction f de l'introduction. On obtient une relation d'équivalence sur le type des suites de nombres rationnels : nos nombres réels constructifs sont un setoïde, que nous avons nommé `CReal`. De même que les nombres rationnels de la bibliothèque standard, `CReal` instancie la classe de types `RelationClasses.Equivalence`[2] et nous utilisons les tactiques correspondantes, `rewrite` et `setoid_replace`.

Les opérations d'addition (`CReal_plus`), soustraction (`CReal_minus`), multiplication (`CReal_mult`) et division (`CReal_inv`) sont définies sur les suites de Cauchy, puis prouvées compatibles avec la relation setoïde, c'est-à-dire qu'elles instancient la classe de types `Morphisms.Proper`. Elles font des nombres réels constructifs un anneau setoïde, via `Add Ring`.

Malheureusement, les nombres réels constructifs ne peuvent pas utiliser la tactique `field`, car leur division est une fonction partielle, qui demande une preuve que le dénominateur n'est pas nul. Ceci est dû à un célèbre théorème de Brouwer [8], d'après lequel toute fonction totale et constructive $\mathbb{R} \to \mathbb{R}$ est continue. Or quelle que soit la valeur arbitrairement choisie pour $\frac{1}{0}$, la fonction inverse ne serait pas continue puisque les limites à gauche et à droite de zéro sont les infinités négative et positive. De nos jours, une preuve du théorème de Brouwer passe plutôt par la production d'un topos (un modèle des mathématiques constructives) dans lequel toute fonction totale $\mathbb{R} \to \mathbb{R}$ est continue. Voir par exemple MacLane et Moerdijk [14]. On pourrait imaginer une variante de la tactique `field`, qui fonctionnerait avec une division partielle, avec preuve de non-nullité du dénominateur.

Revenons enfin sur la canonicité et le calcul. Le fichier `Coq.Reals.ConstructiveRcomplete` définit sans axiome la fonction

```
Definition RQ_limit : forall (x : CReal) (n:nat),
  { q:Q  &  x < inject_Q q < x + inject_Q (1 # Pos.of_nat n) }.
```

La notation `{q:Q & ...}` se lit "il existe un nombre rationnel q tel que ...", c'est un sigma-type. `inject_Q` est l'injection canonique de $\mathbb{Q}$ dans $\mathbb{R}$. Ainsi `let (q,_) := RQ_limit x n in q` est un nombre rationnel qui approxime `x` à $\frac{1}{n}$ près; on peut demander à Coq de mettre ce lambda-terme en forme normale, par la commande
`Compute (let (q,_) := RQ_limit x n in q)`. Par canonicité on obtiendra deux suites de bits explicites, une pour le numérateur et une pour le dénominateur. Au passage, l'implémentation de `RQ_limit` est très simple : il suffit d'évaluer la suite de Cauchy de x en un indice de convergence à $\frac{1}{n}$ près.

[2] voir le manuel de Coq [11] pour toutes les références techniques citées

2.2 Classiques

Pour obtenir des nombres réels non calculables, il suffit de décréter que les suites de Specker en sont. Ces suites rationnelles sont croissantes et majorées, ce qui peut suffire à convaincre qu'elles convergent vers des nombres. Ici on souhaite davantage : un type $\mathbb{R}$ structuré, notamment avec des opérations algébriques. On choisit donc d'ajouter 3 axiomes logiques, ce qui préserve les propriétés de $\mathbb{R}$ démontrées constructivement, afin de prouver la convergence (non calculable) des suites de Specker. Les voici

- `sig_forall_dec` : le principe d'omniscience limitée, qui tranche si une suite booléenne atteint la valeur vraie ou non ;
- `sig_not_dec` : le tiers exclu pour les négations ;
- `functional_extensionality_dep` : égalité de deux fonctions si leurs valeurs sont égales en chaque argument.

Le principal intérêt de ces 3 axiomes est qu'ils sont vrais dans les modèles ensemblistes de Coq [13][17]. Par conséquent ils sont consistents, ce qui n'était pas prouvé, à notre connaissance, pour les précédents axiomes ad hoc des nombres réels. Concernant leur force, les deux premiers étaient des théorèmes dans l'axiomatique précédente, comme on le voit dans le fichier `Coq.Reals.Rlogic`, donc on ne paye pas plus cher qu'avant. Nous avons ajouté le troisième, l'extensionalité fonctionnelle (funext), car nous en avons besoin pour effectuer le quotient du setoïde des nombres réels constructifs, afin d'avoir l'égalité de Leibniz sur les nombres réels classiques. Funext est généralement considérée comme assez modeste logiquement, elle est par exemple postulée dans la théorie des types homotopiques [18].

En présence de ces 3 axiomes logiques, nous changeons d'approche, et définissons les nombres réels classiques par les coupures de Dedekind. Ce sont les intervalles de nombres rationnels infinis à gauche et ouverts à droite, qui définissent un nombre réel par cette limite à droite

```
Definition isLowerCut (f : Q -> bool) : Prop
  := (forall q r:Q, q < r -> f r = true -> f q = true)   (* interval *)
     /\ not (forall q:Q, f q = true)      (* avoid positive infinity *)
     /\ not (forall q:Q, f q = false)     (* avoid negative infinity *)
     /\ (forall q:Q, f q = true
            -> not (forall r:Q, r <= q \/ f r = false)). (* openness *)
Definition DReal : Set
  := { f : Q -> bool | isLowerCut f }.
```

Par exemple $\{q \in \mathbb{Q} \ / \ q < 0 \text{ ou } q^2 < 2\}$ est la coupure de Dedekind qui représente le nombre réel $\sqrt{2}$. Nous avons réimplémenté le type `R` de la bibliothèque standard, anciennement axiome, comme un alias de `DReal`.

Funext prouve que `DedekindReal` a l'égalité classique attendue : deux coupures de Dedekind sont égales si et seulement si elles ont la même valeur en chaque nombre rationnel. Notons que cette égalité requiert aussi que les preuves de `isLowerCut` soient égales, c'est-à-dire que `isLowerCut f` soit une H-Prop, un type singleton ou vide. Ceci est de nouveau démontré par funext, d'après qui toute négation est une H-Prop, toute conjonction de H-Props est une H-Prop, et la clôture universelle de H-Props est une H-Prop. Voilà pourquoi nous avons formulé `isLowerCut` par des négations.

En définissant l'ordre large $\leq$ des réels de Dedekind par l'inclusion des intervalles rationnels, ceci entraîne immédiatement

```
Lemma Rle_antisym : forall x y : DReal,
    x <= y  ->  y <= x  ->  x = y.
```

L'équivalence setoïde des réels constructifs devient l'égalité de Leibniz dans les réels classiques de Dedekind, ce qui était un de nos objectifs. L'autre propriété classique attendue est la totalité de l'ordre sur les nombres réels

```
Lemma total_order_T :
  forall x y:R, {x < y} + {x = y} + {y < x}.
```

Ce terme était un axiome dans la précédente axiomatique. Voyons comment nous l'avons démontré sur les réels de Dedekind classiques. Soit x, y deux tels réels. On encadre x par deux rationnels, $a < x \leq b$, ce qui veut dire que la coupure de Dedekind x vaut true sur a et false sur b. On évalue alors la coupure y sur a et b. Si une valeur est différente, on en conclut immédiatement que $x < y$ ou que $y < x$. Il reste le cas $a < y \leq b$. Alors on construit par dichotomie une suite rationnelle (a_n) qui converge vers x strictement par en dessous : si $\frac{a+b}{2} < x$ alors un encadrement 2 fois plus fin est $\frac{a+b}{2} < x \leq b$, sinon on prend $a < x \leq \frac{a+b}{2}$ et ainsi de suite. On forme ensuite la suite booléenne (b_n) des évaluations de la coupure y sur (a_n). Le principe d'omniscience limitée tranche si (b_n) touche la valeur false, auquel cas on conlut que $y < x$. Sinon on a $x \leq y$ et le même raisonnement en inversant x et y permet de trancher entre $x = y$ et $x < y$.

Le dernier axiome de la précédente bibliothèque standard était `completeness`, l'existence de bornes supérieures pour toute partie `E : R->Prop` non vide et majorée. On remarque que la proposition de majoration de `E` est négative

```
Definition is_upper_bound (E:R -> Prop) (m:R)
  := forall x:R, E x -> x <= m.
```

car elle est équivalente à `not exists x:R, E x /\ m < x`. Elle est donc tranchée par l'axiome `sig_not_dec`, ce qui nous a permis de construire par dichotomie une suite de nombres rationnels qui converge vers la borne supérieure de `E`.

Nous avons finalement relié les nombres réels constructifs et classiques par une opération de quotient. C'est-à-dire que nous avons défini une surjection

```
Definition Rabst : CReal -> DReal.
Definition Rrepr : DReal -> CReal.
Definition Rquot1 : forall x y:DReal, Req (Rrepr x) (Rrepr y) -> x = y.
Definition Rquot2 : forall x:CReal, Req (Rrepr (Rabst x)) x.
```

où `Req` est l'équivalence setoïde des réels constructifs. La fonction `Rrepr` est la construction d'une suite de Cauchy rationnelle qui converge vers la limite d'une coupure de Dedekind, nous venons d'en parler lors de la démonstration de la totalité de l'ordre. La fonction `Rabst` demande de décider dans les booléens si un rationnel a est strictement plus petit que la limite d'une suite de Cauchy rationnelle (r_n), ce qui est tranché par l'omniscience limitée.

3 Inclusion du constructif dans le classique

Les nombres réels et l'analyse classiques sont une extension logique de leurs versions constructives, en ajoutant les 3 axiomes `sig_forall_dec`, `sig_not_dec` et `functional_extensionality_dep`. Il n'y a donc pas lieu d'opposer ces deux pratiques des mathématiques, il vaut mieux chercher comment appeler les théorèmes constructifs depuis les théorèmes classiques.

Le premier exemple de partage de preuves se voit dans les opérations algébriques des nombres réels, les termes `R0, R1, Rplus, Rmult, Ropp, Rinv, Rlt`. Axiomes dans la précédente bibliothèque standard, nous avons dû les implémenter[3] pour les réels de Dedekind classiques. Les définitions et preuves que les réels de Dedekind forment un anneau sont possibles en manipulant directement les coupures de Dedekind, nous l'avons fait en partie dans ce dépôt [2], cela prend environ 2000 lignes de Coq. Pour la bibliothèque standard, puisque nous avons déjà fait ce travail pour `CReal`, nous avons trouvé plus intéressant de poser

```
Definition Rmult : R -> R -> R
  := fun x y : R => Rabst (CReal_mult (Rrepr x) (Rrepr y)).

Lemma Rmult_eq_reg_l : forall r r1 r2, r * r1 = r * r2 -> r <> 0 -> r1 = r2.
Proof.
  intros. apply Rquot1. apply (Rmult_eq_reg_l (Rrepr r)).
  rewrite <- Rrepr_mult, <- Rrepr_mult, H. reflexivity.
  rewrite Rrepr_appart, Rrepr_0 in H0. exact H0.
Qed.
```

Dans ces deux cas, on utilise le quotient `Rabst, Rrepr` pour traduire le problème de Dedekind vers Cauchy, puis résoudre le problème dans Cauchy et revenir dans Dedekind. Ce n'est donc pas complètement automatique, mais la partie classique des opérations algébriques réelles passe ainsi de 2000 à 100 lignes de Coq.

Autrement, nous proposons d'interpréter `Rabst, Rrepr` comme un isomorphisme entre `CReal` et `DReal`. À cette fin nous avons défini, à l'instar de C-CoRN, une interface de nombres réels constructifs nommée `ConstructiveReals`, et nous avons prouvé que `CReal` et `DReal` en sont 2 implémentations différentes. Ensuite nous avons défini les morphismes `ConstructiveRealsMorphism`, pour enfin prouver que `Rabst, Rrepr` constitue un isomorphisme. Ceci donne une nouvelle option pour partager des théorèmes constructifs : les démontrer avec quantificateur universel sur les implémentations de `ConstructiveReals`, ce qui est presque aussi facile que de travailler dans `CReal`. On n'a alors même plus besoin de `Rabst, Rrepr`, il suffit d'instancier les théorèmes sur `CReal` ou `DReal`, notamment en les déclarant comme `Canonical Structure`.

Au passage nous avons vérifié informellement que `ConstructiveReals` est isomorphe à l'interface de C-CoRN. Nous pensons que c'est une des missions de la bibliothèque standard que de rassembler ces interfaces et définitions de concepts mathématiques de base. Si les bibliothèques externes les implémentent, leurs travaux deviendront inter-opérables. Il reste une question lorsque les théorèmes ne s'énoncent pas pareil en constructif et en classique, nous y reviendrons dans la section Travail futur.

[3] Pour la rétro-compatibilité nous avons dû les garder opaques, par des Module Type implémentés de façon opaque, cf fichier Coq.Reals.Rdefinitions.

4 Performance

Rappelons que la motivation première des mathématiques constructives n'est pas la vitesse des calculs, mais la vérité. Voir par exemple l'introduction de [4]. Pour un constructiviste, la donnée d'une règle de calcul est le seul moyen de prouver l'existence d'un objet mathématique. De notre opinion, un objet mathématique n'est pas un pot de fleurs qu'on pourrait localiser dans une cuisine, le lieu de son existence est problématique. Nous pensons que les mathématiques constructives résolvent ce problème : un objet constructif existe au bout de son calcul. Autrement dit, un objet existe parce que je peux le construire.

La question de la vitesse des calculs arrive dans un deuxième temps pour un constructiviste, nous en parlons brièvement dans cette section. Pour l'instant, les réels constructifs que nous avons introduits dans la bibliothèque standard sont horriblement lents, inaptes à toute application pratique, encore moins industrielle. C'est-à-dire que la bibliothèque standard permet désormais de définir sans axiome la fonction exponentielle, et de prouver ses propriétés. En revanche, la mise sous normale du terme `let (q,_) := RQ_limit (exp 1) 10 in q` est un calcul qui ne termine pas en un temps raisonnable.

Ceci étant nous avons prouvé une forte propriété de notre interface `ConstructiveReals` : toutes ses implémentations sont isomorphes et les isomorphismes sont extensionnellement uniques. On pourra donc ajouter des implémentations plus tard, et faire passer les calculs de l'une à l'autre par isomorphisme, en choisissant la plus rapide pour chaque étape des calculs.

5 Travail futur

Suite au working group de Coq à Nantes le 1er octobre 2019, nous avons discuté de l'opportunité de retirer les preuves d'analyse réelle de la bibliothèque standard, pour n'y laisser que les interfaces et définitions de base, rapidement évoquées dans la section précédente. Les preuves seraient déplacées dans des bibliothèques existantes telles que Coquelicot, math-comp/analysis et C-CoRN. Ceci éviterait des redondances entre ces bibliothèques, et simplifierait l'accès à l'analyse réelle en Coq. Cela permettrait aussi de simplifier les preuves des bibliothèques classiques si, via l'interface commune de la bibliothèque standard, des preuves de math-comp appelaient par exemple des preuves de C-CoRN.

Revenons maintenant aux théorèmes d'analyse réelle qui ne s'expriment pas pareil en constructif et en classique, par exemple le théorème des valeurs intermédiaires (TVI). En mathématiques classiques ce théorème s'énonce : pour toute fonction continue $f : [a, b] \to \mathbb{R}$ telle que $f(a) < 0 < f(b)$, il existe un nombre réel $c \in]a, b[$ tel que $f(c) = 0$. Constructivement ce théorème a plusieurs variantes. Puisque la logique constructive est plus faible, nous pouvons ou bien affaiblir la conclusion du TVI, ou bien renforcer ses hypothèses. Ici nous prenons la seconde option et demandons de surcroît que la fonction f soit localement non-nulle : pour tout $x \in [a, b]$ et $m \in \mathbb{N}^*$, il existe $y \in [a, b]$ tel que $|x - y| < \frac{1}{m}$ et $f(y) \neq 0$. Cette hypothèse est assez souvent satisfaite, notamment par les fonctions analytiques. Et avec elle, le TVI se prouve constructivement. Pour en conclure la version classique du TVI, il suffit d'un coup de tiers exclu sur la non-nullité locale de f.

Nous passons enfin à l'échelle de toute une théorie, à savoir la théorie de la mesure et de l'intégrale de Lebesgue [7]. Cette fois-ci c'est la version classique de la théorie qui a plusieurs variantes, équivalentes, notamment l'intégrale de Daniell [9]. Sans rentrer dans les détails, disons seulement que Bishop et Cheng [5] ont montré que c'est la variante de Daniell qui est

constructive. C'est-à-dire, grossièrement, que les preuves de Daniell repoussent les usages du tiers exclu très tard, à l'instar de ce qu'on vient de voir pour le TVI. Par conséquent, on peut espérer réduire considérablement les preuves de l'intégrale classique : séparer dans Daniell la quasi-totalité du travail qui est constructif (ce qu'ont fait Bishop et Cheng), ne laisser dans Daniell classique que des appels au tiers exclu par dessus des appels à Daniell constructif, enfin réduire Lebesgue à des appels à Daniell classique.

Remerciements. Nous remercions chaleureusement Bas Spitters, Hugo Herbelin et Guillaume Melquiond, qui au cours de nos dicussions ont fourni la plupart des idées implémentées ici, et sans qui l'intégration de ce travail dans Coq n'aurait pas été possible.

References

[1] The coq constructive repository at nijmegen. https://github.com/coq-community/corn.

[2] Andrej Bauer. Dedekind reals. https://github.com/andrejbauer/dedekind-reals.

[3] Yves Berthot and Guillaume Allais. Views of pi: definition and computation, 2014.

[4] Errett Bishop. Foundations of constructive analysis. Ishi press international, 2012.

[5] Errett Bishop and Henry Cheng. Constructive measure theory. Memoirs of the American mathematical society, Number 116, 1972.

[6] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. https://hal.inria.fr/hal-00806920v1/document, 2013.

[7] Nicolas Bourbaki. Éléments de mathématique, intégration, 1967.

[8] Luitzen Egbertus Jan Brouwer. Über definitionsbereiche von funktionen. Mathematische Annalen, 97: 60–75, 1927.

[9] Percy John Daniell. A general form of integral. Annals of Mathematics. Second Series., 1918.

[10] Herman Geuvers and Milad Niqui. Constructive reals in coq: Axioms and categoricity, 2000.

[11] Inria. La documentation de coq. https://coq.inria.fr/doc.

[12] Robbert Krebbers and Bas Spitters. Computer certified efficient exact reals in coq. https://arxiv.org/abs/1105.2751, 2011.

[13] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of cc with predicative induction and judgmental equality. https://arxiv.org/abs/1111.0123, 2011.

[14] Saunders MacLane and Ieke Moerdijk. Sheaves in geometry and logic. Springer Verlag, 1992.

[15] Russell O'Connor. A monadic, functional implementation of real numbers, 2007.

[16] Ernst Specker. Nicht konstruktiv beweisbare sätze der analysis. J. Sym. Logic, vol. 14, no 3, 1949.

[17] Amin Timany and Matthieu Sozeau. Consistency of the predicative calculus of cumulative inductive constructions (pcuic). https://arxiv.org/abs/1710.03912, 2017.

[18] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Better Automation for TLA$^+$ Proofs

(Work in progress)

Antoine Defourné

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

Abstract

TLA$^+$ is a specification language based on traditional untyped set theory. It is equipped with a set of tools, including the TLA$^+$ proof system TLAPS, which uses trusted back-end solvers to handle individual proof steps—referred to as "proof obligations". As most solvers rely on and benefit from *typed* formalisms, types are first reconstructed for the obligations; however, the current encoding into the SMT-LIB format does not exploit all of this type information. In this paper, we present motivations for a more pervasive usage of types at an intermediate representation of TLA$^+$ proof obligations, and describe work in progress on several improvements of TLAPS: a type-driven SMT encoding, a tactic for instantiation hints, and type annotations for the language. We conclude with some perspectives for future work.

1 Introduction

TLA$^+$ is a formal specification language designed by Leslie Lamport [Lam02]. It is used to write system specifications, in particular of concurrent systems. It is based on the *Temporal Logic of Actions* (TLA)—an extension of first-order logic (FOL) with a notion of state and temporal operators—combined with the operators of *untyped set theory*. This choice of traditional set theory in first-order logic makes TLA$^+$ very similar to the language of usual mathematics, but the absence of types is uncommon among practical formalisms. This design principle is discussed in a paper by Lamport and Paulson [LP99]; it offers flexibility, expressiveness, and ease of use for the system designer.

Other systems based on (typed) set theory include the Z language [Spi92], Event-B and the Rodin platform [Abr10] [DFGV14], and the Mizar proof assistant [KP19]. Systems based on untyped set theory are not as many, but Isabelle/ZF is an example [Pau93].

On the left-hand side of figure 1 is an example of a TLA$^+$ specification. This code specifies sorting algorithms at an abstract level; it leaves out the details of potential implementations, and it could be *refined* by a more concrete specification [HHK$^+$15]. Several constants are first declared, plus one variable `arr`, which represents the array to sort. It is initially equal to `arrInit`, as stated in the predicate `Init`. The predicate `Next` contains the primed variable `arr'` representing the array *in the next state*; that makes `Next` an *action*, that is a predicate expressing a relation between two consecutive states. Finally, the predicate `Spec` assembles the initial predicate and the action into one *temporal formula*. Here the box modality `[]` prefixes a formula that must hold true across all states. `[][Next]_arr` is an abbreviation for `[](Next \/ arr' = arr)`. Thus `Spec` states that "`Init` is initially true, and for all consecutive states either `Next` is true, or the variable `arr` does not change."

TLA$^+$ comes with a set of tools, among which are the finite model-checker TLC [YML99] and the proof system TLAPS[1] [CDL$^+$12]. TLC is useful to simulate finite behaviors (sequences of states) of a specification and check invariants and temporal properties. For example, we can test that the termination of the algorithm implies the array is sorted, by verifying with TLC

[1] Developed by the Microsoft Research–INRIA Joint Centre.

```
(* N: size of the array;
 * Max: maximal value;
 * arrInit: the initial array
 * arr: the array to sort *)
CONSTANTS N, Max, arrInit
VARIABLE arr

Dom == 1 .. N
USE DEF Dom

ASSUME NIsNat == N ∈Nat
ASSUME MaxIsNat == Max ∈Nat
ASSUME ArrInitIsArray ==
       arrInit ∈ [ Dom -> 1 .. Max ]

Init == arr = arrInit

Swap(f, i, j) == [ f EXCEPT ![i] = f[j],
                            ![j] = f[i] ]

Next == ∃ i, j ∈ Dom :
          /\ i < j
          /\ arr[i] > arr[j]
          /\ arr' = Swap(arr, i, j)

Spec == Init /\ [][Next]_arr

Sorted(f) == ∀ x, y ∈ DOMAIN f :
               x < y => f[x] <= f[y]
```

```
LEMMA PermId ==
    ∀ f ∈ [ Dom -> 1 .. Max ] : IsPermOf(f, f)

LEMMA PermComp ==
    ∀ f, g ∈ [ Dom -> 1 .. Max ] :
        ∀ i, j ∈ Dom :
            IsPermOf(f, g) => IsPermOf(Swap(f, i, j), g)

THEOREM Spec => []IsPermOf(arr, arrInit)
<1>1 Init => IsPermOf(arr, arrInit)
    <2> HAVE Init
    <2> SUFFICES IsPermOf(arrInit, arrInit)
        BY DEF Init
    <2> QED
        BY PermId, ArrInitIsArray
<1>2 IsPermOf(arr, arrInit) /\ Next
        => IsPermOf(arr', arrInit)
    <2> SUFFICES ASSUME IsPermOf(arr, arrInit),
                        NEW i ∈Dom,
                        NEW j ∈Dom
                 PROVE IsPermOf(Swap(arr, i, j), arrInit)
        BY DEF Next
    <2> QED
        BY PermComp, ArrInitIsArray
<1>3 IsPermOf(arr, arrInit) /\ arr' = arr
        => IsPermOf(arr', arrInit)
    OBVIOUS
<1> QED
    BY PTL, <1>1, <1>2, <1>3 DEF Spec
```

Figure 1: Example of TLA$^+$ Specification and Proof

that the TLA$^+$ expression "`(~ ENABLED Next) => Sorted(arr)`" is an invariant. However this requires the user to provide the parameters N, Max and arrInit. To verify properties for all parameter values, TLC is insufficient since the search space is infinite, thus we need to write a formal proof with TLAPS. This tool is able to parse a TLA$^+$ *proof script* and generate several *proof obligations* for it; each obligation corresponds roughly to an individual proof step, and must be checked by one of the trusted back-end solvers available.

The right-hand side of figure 1 contains a proof script. Here a different property of the specification is proved: the array is always a permutation of the initial array.[2] Proofs in TLAPS are hierarchical in the sense that each proof is either one line of relevant facts and definitions, or a sequence of steps, each step justified by its own proof. We have left hidden a few parts of the module, such as the definition of IsPermOf, and proofs of lemmas PermId and PermComp, in order to lighten the presentation. The structure of this proof is simple and follows a very common pattern: first it is shown that the property holds in the initial state; then that it is preserved by the action (two values are swapped); then that it is also preserved when the array is unchanged; finally, all proof steps are assembled to prove the main goal, as indicated by the keyword QED. The particular method PTL is invoked here to reason about temporal formulas.

2 Overview of the SMT Encoding

In previous work [Van14], Hernán Vanzetto defined an encoding of proof obligations into SMT-LIB, the standard input format for SMT solvers [BFT17]. This encoding underlies the interface

[2] This is an example of a "safety property"; it states that some invariant is maintained by the specification. Properties about an interesting state being eventually reached (like "the array is eventually sorted") are called "liveness properties".

of TLAPS with SMT solvers such as CVC4, Z3 and VeriT. In this section we sum up the important points of this encoding.

The language of SMT-LIB is based on multi-sorted first-order logic (MS-FOL) with equality; the encoding of TLA$^+$ can thus be seen as a translation of set theory into that logic. The encoding is untyped in the sense that most expressions in the source problem are translated to terms of a fixed sort U (or *bool* when a formula is expected). The translation proceeds in two steps: first, the source obligation is transformed into an equivalent one which is free of so-called non-basic expressions—that is all expressions that do not have a counterpart in the target logic, like set theoretic expressions—then, the basic obligation is encoded into SMT-LIB in a straightforward way.

Several techniques are at play to reduce a TLA$^+$ formula to a basic form. One of them is to apply rewriting rules, such as

$$x \in A \cap B \longrightarrow x \in A \wedge x \in B$$

which eliminates non-basic expressions (in that case $A \cap B$). There are cases when a non-basic expression appears in a context where no rewriting rule can be applied; in that case the expression is replaced by a fresh constructor. This technique was named "abstraction" in [Van14], here is an example:

$$A \cap B \in S \overset{\mathbf{k}\ \text{fresh}}{\longrightarrow} \big(\forall a \forall b.\, \mathbf{k}(a,b) = a \cap b\big) \wedge \mathbf{k}(A,B) \in S$$

When the abstracted non-basic expression is higher-order, the fresh constructor must be parameterized by the free variables of the original expression. For example:

$$P\big(\{x \in S : \phi(\vec{y}, x)\}\big) \overset{\mathbf{k}\ \text{fresh}}{\longrightarrow} \big(\forall s \forall \vec{y}.\, \mathbf{k}(s,\vec{y}) = \{x \in s : \phi(\vec{y}, x)\}\big) \wedge P\big(\mathbf{k}(S,\vec{y})\big)$$

Using these core techniques it is possible to handle most TLA$^+$ expressions. The remaining expressions are the ones involving specialized reasoning, like arithmetic for example. Consider the valid TLA$^+$ formula

$$\forall x.\, x \in \mathit{Int} \Rightarrow x + 0 = x$$

As the source language is untyped, the default way to encode this formula would be to annotate x with the type U, encode addition with an operator $+_U$ of signature $\langle U \times U \rangle \to U$, and add axioms that specify the behavior of $+_U$. A major drawback of this approach is that it prevents the solver to make use of its procedures to reason about arithmetic expressions.

We can partially remedy this problem by linking the specialized sort *int* with the default sort U, using an injective "cast" operator ${}^{\mathit{int}}_{U}{\downarrow}\cdot$ of type $\mathit{int} \to U$. We add the following axioms in the target problem:

$$\forall x^U.\, x \in \mathit{Int} \Leftrightarrow \exists n^{\mathit{int}}.\, x = {}^{\mathit{int}}_{U}{\downarrow}n$$
$$\forall m^{\mathit{int}} \forall n^{\mathit{int}}.\, \big({}^{\mathit{int}}_{U}{\downarrow}m\big) +_U \big({}^{\mathit{int}}_{U}{\downarrow}n\big) = {}^{\mathit{int}}_{U}{\downarrow}(m +_{\mathit{int}} n)$$

And translate the formula into $\forall x^U.\, x \in \mathit{Int} \Rightarrow x +_U \big({}^{\mathit{int}}_{U}{\downarrow}0\big) = x$. Then there is no more need to axiomatize $+_U$ explicitly. However, the resulting problem is obsfuscated with additional casts and quantifiers.

This issue is presented in [Van14] as a motivation for the introduction of *reconstructed types*. Before an obligation is encoded, an algorithm attempts to infer types to annotate bound variables. If this procedure succeeds, is becomes possible to optimize the encoding. An elementary type system for TLA$^+$ comprises atomic types, functional types $\tau_1 \to \tau_2$ and set types $\mathrm{Set}(\tau)$.

In our example, the type *int* can be inferred for x, which results in the hypothesis $x \in Int$ being simplified, and the arithmetical expression to be encoded directly:

$$\forall x^{int}.\, x +_{int} 0 = x$$

The type reconstruction phase may fail, in that case TLAPS falls back on the untyped encoding. Currently TLAPS implements a rich type system with dependent types and refinement types, which allow for even finer optimizations.

3 Towards a Better Encoding of Untyped Set Theory

In this section we present a series of potential improvements of the current encoding of TLA$^+$ into SMT-LIB. All of these ideas revolve around types, and are somewhat motivated by the assumption that SMT solvers perform best with types in general. Although type reconstruction is performed, the encoding we described above does not harness the full potential of this extra type information. It is used for optimizations involving objects of an atomic type, like integers, or to check conditions like membership in some function's domain, but by default the encoding remains untyped.

Type-driven Encoding

Consider the following theorem of TLAPS:

```
THEOREM ASSUME NEW f ∈[ Int -> Int ]
        PROVE ∀z ∈ Int : f[z] ∈ Int
    OBVIOUS
```

This theorem leads to a single proof obligation, for which type reconstruction leads to the typing $f : int \to int$ and $z : int$. The (simplified) SMT-LIB translation of the obligation is:

```
(declare-sort U 0) ;; represents the universe of sets

(declare-fun int2u (Int) U)
(declare-fun intSet () U)       ;; Int : U
(declare-fun mem (U U) Bool) ;; \in : U -> U -> bool
(declare-fun domain (U) U)    ;; DOMAIN : U -> U
(declare-fun app (U U) U)     ;; _[_] : U -> U -> U

(declare-fun f () U)
(declare-fun z () Int)

(assert (not (mem (app f (int2u z)) intSet))) ;; goal
(assert (= (domain f) intSet))
(assert ( forall ((v Int)) (mem (app f (int2u v)) intSet)))
```

(The two last assertions are axioms resulting from the hypothesis `f ∈ [ Int -> Int ]`.) As we can see the type of z actually resulted in the declaration of the constant `z` of sort `Int`; however the type of f was simply discarded, and casts from `Int` inserted to make the expressions well-typed. This is coherent with the motivation behind type reconstruction: types merely serve to recognize expressions of a specific domain (like integers) and perform small optimizations ("$x \in Int$" is always true when $x : int$).

Our proposal is that an encoding should make use of *all* the types, including functional types and set-types. Following [Ker91] and [Bla12], we propose to extend the encoding so as to map each original type to a corresponding sort in SMT-LIB. The result file could look like this:

```
(declare-sort ArrowIntInt 0)  ;; represents 'int -> int'
(declare-sort SetInt 0)       ;; represents 'Set(int)'

(declare-fun intSet () SetInt)              ;; Int : Set (int)
(declare-fun mem (Int SetInt) Bool)         ;; \in : int -> Set(int) -> bool
(declare-fun domain (ArrowIntInt) SetInt)   ;; DOMAIN : (int -> int) -> Set(int)
(declare-fun app (ArrowIntInt Int) Int)     ;; _[_] : (int -> int) -> int -> int

(declare-fun f () ArrowIntInt)  ;; f : int -> int
(declare-fun z () Int)          ;; z : int

(assert (not (mem (app f z) intSet))) ;; goal
(assert (= (domain f) intSet))
(assert (forall ((v Int)) (mem (app f v) intSet)))
```

The sort ArrowIntInt corresponds to the type $int \to int$, and the sort SetInt to the type Set(Int), which is the type of the constant Int in TLA$^+$. As we can see it is possible to get rid of the generic sort U, as well as cast functions.

Of course, operators like set membership and application cannot be attached a unique sort, so duplicates of mem, app, etc. will have to be declared for all cases. We intend to implement this extension for the encoding using elementary types, since it is not clear yet if dependent and refinement types could be encoded easily in a similar way. With the universal domain U eliminated and replaced by a plurality of more specialized domains, we expect this encoding to facilitate instantiation, and improve the overall performance of the SMT backends.

Instantiation Hints

It is well-known that instantiation of quantifiers is difficult for SMT solvers. As TLAPS relies solely on its trusted back-ends, it can become a problem if an obligation requires difficult instantiations—in fact even simple ones sometimes lead to failure.

We propose to exploit a feature of SMT-LIB [BFT17, p. 31] to remedy such problems. In SMT-LIB, quantifiers may be annotated with patterns, which are formulas that may trigger an instantiation when a match is detected. Formally, if we consider a formula

$$\forall x_1 \ldots \forall x_n.\, \phi(a_1, \ldots, a_m, x_1, \ldots, x_n)$$

then a pattern for this formula is a list of terms $(t_i)_{1 \leq i \leq p}$ such that the free variables of each t_i are contained in $\{a_1, \ldots, a_m, x_1, \ldots, x_n\}$. A match for such a pattern is a substitution θ of the variables such that all terms $t_i\theta$ appear somewhere. The purpose of a pattern is to suggest that whenever a match θ is found by the SMT solver, the formula should be instantiated with the terms $\theta(x_1), \ldots, \theta(x_n)$. There may be several patterns for one quantified formula.

Suppose we need to prove a goal that involves instantiating a quantifier with some known expression e. This can happen if the goal is existential, or some hypothesis is universal, for instance. We can do this by declaring a fresh unary predicate W (for "with", or "witness") in the encoded SMT-LIB problem, add the axiom "$W(e)$", and the pattern consisting of the single formula "$W(x)$" for every quantifier Qx that requires to be instantiated. Although doing

so introduces more patterns than necessary, especially if one single quantifier is aimed for, we believe that many unnecessary instantiations will still lead to better performances than no instantiations at all.

In the context of an untyped encoding, there is no difficulty; an instantiation hint w can be encoded as a term w' of type U. But if the encoding is typed, like the one discussed in the previous section, several things must be arranged. First, there should be a different predicate W_τ for each type τ (as many as necessary). Second, the pattern $W_\tau(x)$ can only annotate quantifiers of the appropriate type Qx^τ; this in fact illustrates a benefit of types, because there are less quantifiers to instantiate for each hint. Third, since w is not part of a larger expression, its type must be computed *after* the type reconstruction phase for the main obligation. This may be impossible in some cases, like in the following problem:

```
Id(S) == [ x ∈ S |-> x ]

THEOREM ASSUME NEW T
        PROVE ∃f ∈ [ T -> T ] : ∀x ∈ T : f[x] = x
   WITH Id(T)
```

The use of the keyword WITH here is not part of the current TLA$^+$ proof language, but it illustrates how the feature could be implemented in the syntax. Here the issue arises from the fact that no annotations are generated for the term $\mathrm{Id}(T)$, because it does not appear in the generated obligation. This problem could be solved if types were generated for all defined operators in the whole module, which is the subject of the next section. It is also possible that this problem does not arise in realistic cases—in our example, we forgot to expand the definition of Id, which is in fact necessary, and actually solves the typing problem because $[x \in T \mapsto x]$ has type $\alpha \to \alpha$ when T has type $\mathrm{Set}(\alpha)$.

Type Annotations

Type reconstruction is currently performed on proof obligations, which are themselves generated from a list of usable facts and definitions. From a single proof script, many obligations can be generated, and all do not share the same amount of information; facts established at some point in a proof may be invoked later, definitions may be hidden or expanded, etc. The BY and DEF keywords control what facts and definitions should be considered usable in a proof step. The purpose of this simple mechanism is to ensure that obligations do not get saturated with useless information. One major downside is that it is very easy to forget necessary facts that seem irrelevant in a proof. In our experience, this happens quite often with simple facts of the form $x \in S$. For example, $n+0 = n$ is only true in TLA$^+$ if the fact $n \in \mathit{Int}$ is visible. Unfortunately in many situations n will be known to belong to some other set S, and $S \subseteq \mathit{Int}$ can only be proven if enough definitions are expanded.

This observation led us to the conclusion that TLAPS could benefit from type annotations at the *module* level. Instead of reconstructing types for each individual obligation, declared operators could be annotated at the top-level, and this information shared for all obligations. We believe this extension will enable new optimizations; returning to our example, if S was an operator defined as $\{n \in \mathit{Nat} : n > 0\}$, S could be attached the type $\mathrm{Set}(\mathit{int})$, and the fact $S \subseteq \mathit{Int}$ could be inferred without expanding the definition of S.

As a next step, we also intend to allow the user to provide his own type annotations when declaring operators. A syntax for this feature could be:

```
IsZero(n : int, f : int -> int) : bool == f[n] = 0
```

IsZero would be given the type $int \times (int \to int) \to bool$.

However, it remains to decide what should be the precise semantics of these annotations, since our intention is not to restrict the class of accepted TLA$^+$ expressions. What should be done if the definition of the operator does not type-check? What if the operator is used with arguments of the wrong type, or in a wrong context, like for instance in the expression $\mathsf{IsZero}(\emptyset, [n \in Nat \mapsto 3.14])$?

4 Perspectives and Conclusion

We expect these various improvements to result in better performance of the SMT back-ends, and a better ergonomy for TLAPS. However, as it was assumed all along that types for the obligations could be reconstructed, it remains to decide what should be done when this procedure fails. The legitimacy of so-called silly expressions is a part of TLA$^+$'s design. More importantly, a back-end for TLA$^+$ would not be complete if it prevented us from proving *silly theorems*, like for instance:

$$\exists n \in Nat\ \exists x.\, x \in n$$

This is true because, if all natural numbers where equal to $\emptyset$, then all natural numbers would be equal to each other, and we would have $0 = 1$.

There is always the possibility to rely on an untyped encoding of TLA$^+$ when the typed encoding fails. But that might lead to a loss in robustness, since one single anomaly in the obligation would significantly reduce the chances to solve it, or at least the performance of the solvers.

Advanced research in type systems for programming languages may offer interesting alternatives to the current *optional* type system of TLA$^+$. *Soft typing* is an approach that relies on the insertion of explicit run-time checks to retain the expressiveness of dynamically-typed languages [CF91]. More recently, the active field of *gradual typing* combines statically-typed expressions with expressions of a type "unknown" [ST06], [Pet19]. We intend to investigate on these systems, and possibly experiment them in TLA$^+$.

Aside from that, we intend to investigate the possibility of using the reasoning capacities of higher-order solvers directly, notably to improve the support of inductive reasoning. This falls within the project Matryoshka, which aims at extending superposition and SMT solvers with higher-order reasoning in a way that preserves their performances.

5 Acknowledgments

This project receives funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka), and funding from the Région Grand Est. I am thankful to my advisors Stephan Merz, Pascal Fontaine and Jasmin Blanchette for their support and guidance.

References

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[Bla12] Jasmin Christian Blanchette. *Automatic proofs and refutations for higher-order logic*. PhD thesis, Technical University Munich, 2012.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ proofs. *CoRR*, abs/1208.5933, 2012.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 278–292. ACM, 1991.

[DFGV14] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating SMT solvers in Rodin. *Sci. Comput. Program.*, 94:130–143, 2014.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[Ker91] Manfred Kerber. How to prove higher order theorems in first order logic. In John Mylopoulos and Raymond Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*, pages 137–142. Morgan Kaufmann, 1991.

[KP19] Cezary Kaliszyk and Karol Pąk. Semantics of Mizar as an Isabelle object logic. *Journal of Automated Reasoning*, 63(3):557–595, Oct 2019.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[Pau93] Lawrence C. Paulson. Set theory for verification: I. from foundations to functions. *CoRR*, cs.LO/9311103, 1993.

[Pet19] Tommaso Petrucciani. *Polymorphic set-theoretic types for functional languages. (Types ensemblistes polymorphes pour les langages fonctionnels)*. PhD thesis, Sorbonne Paris Cité, France, 2019.

[Spi92] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and functional programming workshop*, pages 81–92, 2006.

[Van14] Hernán Vanzetto. *Proof automation and type synthesis for set theory in the context of TLA+. (Automatisation de preuves et synthèse de types pour la théorie des ensembles dans le contexte de TLA+)*. PhD thesis, University of Lorraine, Nancy, France, 2014.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

Des transformations logiques passent leur certificat

Quentin Garchery[1,2], Chantal Keller[1], Claude Marché[2,1], et Andrei Paskevich[1,2]

[1] LRI, Univ. Paris-Sud, CNRS UMR8623, Orsay, Université Paris-Saclay
Quentin.Garchery@lri.fr
[2] Inria, Université Paris-Saclay, 91120 Palaiseau

Résumé

Dans un contexte de vérification formelle de programmes, utilisant des démonstrateurs automatiques, la *base de confiance* des environnements de vérification est typiquement très large. Ainsi, un outil de vérification de programmes tel que Why3 comporte de nombreuses procédures complexes : génération de conditions de vérification, transformations logiques de tâches de preuve et interactions avec des démonstrateurs externes. En ne considérant que les transformations logiques dans Why3, leur implantation comporte déjà plus de 17000 lignes de code OCaml. Afin d'augmenter notre confiance dans la correction d'un tel outil de vérification, nous proposons un mécanisme de *transformations certifiantes*, produisant des certificats pouvant être validés par un outil externe, selon l'approche *sceptique*. Nous présentons ce mécanisme de génération de certificats et explorons deux méthodes pour les valider : une fondée sur un vérificateur dédié développé en OCaml, l'autre reposant sur le vérificateur de preuves universel Dedukti. Une spécificité de nos certificats est d'être « à petits grains » et composables, ce qui rend notre approche incrémentale, permettant d'ajouter graduellement de nouvelles transformations certifiantes.

1 Introduction

Les outils de vérification de programmes ont pour rôle de garantir qu'un programme vérifie un *contrat* donné, portant par exemple sur des propriétés fonctionnelles du programme, sur l'absence d'erreur à l'exécution, etc. Ces outils se distinguent par l'expressivité des contrats possibles, le paradigme choisi pour établir le contrat, mais aussi par la taille de la *base de confiance*, c'est-à-dire la quantité de code auquel on doit faire confiance pour garantir que les contrats sur les programmes sont bien validés. Un objectif général de tels outils est d'être le plus automatisés possible, et, dans de nombreux cas, cette automatisation se fait à l'aide de code venant s'ajouter à la base de confiance. Un extrême est représenté par Why3 [6], qui permet l'utilisation de dizaines de prouveurs automatiques différents, et leur fait confiance.

Pour réduire la base de confiance de ces outils, une famille d'entre eux se base sur des environnements généralistes d'assistance à la preuve, comme Coq ou Isabelle. Cela reporte la confiance sur celle que l'on a dans l'assistant, qui est lui-même basé sur un noyau que l'on essaye de garder le plus petit possible. Le revers de la médaille est que les preuves doivent se faire obligatoirement dans la logique fournie par l'environnement, et souvent de manière interactive. Pour éviter cet inconvénient, nous présentons dans cet article une approche *sceptique*, qui ne nécessite pas de plonger l'outil de preuve de programmes dans un assistant de preuve. L'idée est d'instrumenter l'outil pour générer un *certificat* de preuve, qui permet de convaincre *a posteriori* de la validité de chaque exécution de l'outil. Ce certificat peut être vérifié par un outil externe qui, lui, peut avoir une petite base de confiance.

Nous avons étudié cette approche dans le cas de l'outil Why3, qui se base sur le paradigme suivant : (1) à partir d'un programme annoté par un contrat et des indications (comme des invariants de boucle), l'outil génère des *obligations de preuve*, à savoir des formules logiques dont la validité entraîne la correction du programme vis-à-vis du contrat ; (2) l'outil cherche à valider ces obligations de preuve, de manière automatique et/ou interactive.

Nous rappelons ci-dessous le processus de preuve de programmes. Il se décompose en quatre grandes étapes. Considérons l'exemple suivant [10] annoté avec des spécifications formelles :

```
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures  { result ≥ 0 }
  = let y = 2*x+1 in a[y*y]
```

1. Le générateur d'obligations produit un ensemble d'énoncés de théorèmes à prouver, appelés *tâches de preuve*. Sur l'exemple, on obtient la formule

```
forall a:array int, x:int.
 length a ≥ 1000 ∧ 0 ≤ x ≤ 10 ∧ (forall i:int.
                    0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0) →
 let y = 2 * x + 1 in (0 ≤ y * y < length a) ∧ a[y * y] ≥ 0
```

2. Chaque tâche de preuve peut être soumise à une ou plusieurs *transformations*, qui produisent un ensemble de sous-tâches à prouver. Sur l'exemple, la transformation `split` produit deux sous-tâches qui sont formées par les deux buts

```
  goal VC1 (* index in array bounds *) : 0 ≤ (y * y) < length a
  goal VC2 (* postcondition *) : a[y * y] ≥ 0
```

dans le contexte commun :

```
a : array int
x : int
Requires1 : length a ≥ 1000 ∧ 0 ≤ x ≤ 10
Requires2 : forall i:int. 0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0
y : int = 2 * x + 1
```

Les transformations sont typiquement déclenchées de manière interactive [10] et sont implantées par du code OCaml interne à Why3. Ainsi, le but `VC2` ci-dessus n'est prouvé par aucun prouveur SMT, à cause de la difficulté d'instancier l'hypothèse `Requires2` avec la bonne valeur de `i`. Un appel manuel de transformation « `instantiate Requires2 x*x+x` » produit la sous-tâche

```
[...] (* meme contexte que ci-dessus *)
Hinst :
  0 ≤ 4 * (x * x + x) + 1 < length a → a[4 * (x * x + x) + 1] ≥ 0

goal VC2 : a[y * y] ≥ 0
```

3. La tâche résultante ci-dessus est maintenant prouvable par un prouveur SMT. Mais afin de faire appel à un prouveur externe donné sur une sous-tâche donnée, d'autres transformations sont appliquées de manière transparente à l'utilisateur afin d'adapter cette sous-tâche à la logique supportée par ce prouveur. Par exemple le type polymorphe `array 'a` est typiquement encodé dans une logique simplement typée.
4. Les sous-tâches obtenues sont transmises aux prouveurs externes et les résultats sont collectés en faisant confiance à la réponse de ces prouveurs.

Toutes ces étapes sont dans la base de confiance de Why3. Le problème auquel on s'attaque est donc le suivant : comment réduire la taille de cette base de confiance ? Dans cet article nous nous focalisons sur la confiance que l'on peut obtenir dans les transformations des étapes 2 et 3. Notre approche est fondée sur l'utilisation de certificats vérifiables par une tierce partie, selon l'approche classiquement dénommée *sceptique*, à l'inverse d'une approche *autarcique* qui chercherait à prouver le code OCaml des transformations. Notre approche s'attache particulièrement à la *modularité* de la certification : nous ne visons pas d'emblée à certifier toutes les étapes ci-dessus, ce qui serait trop ambitieux, mais à ajouter petit à petit des certificats. Nous devons donc insister sur la modularité et la composabilité de nos certificats.

Dans la partie 2 nous présentons la structure de certificats que nous proposons et les propriétés de correction attendues, ainsi que les propriétés de composabilité. La partie 3 présente notre implantation OCaml d'un vérificateur de certificats. Cette implantation OCaml forme ainsi la base de confiance du nouveau mécanisme de transformations certifiantes de Why3. Afin de réduire encore cette base de confiance, nous proposons dans la partie 4 un deuxième vérificateur de certificats qui se base sur la plateforme Dedukti [4]. Dans la section 5 nous présentons des résultats expérimentaux. Le code source associé à ce travail est distribué avec Why3.

2 Certificats modulaires pour les transformations logiques

Nous présentons ici noter méthode pour obtenir des transformations certifiantes. Un premier pas technique est nécessaire pour contourner les contraintes liées à l'API de Why3, décrite brièvement en section 2.1 : afin de nous abstraire de la représentation Why3 des tâches, nous cherchons d'abord, en section 2.2, à extraire le contenu logique de ces tâches. En section 2.3 nous présentons notre langage de certificats et leur sémantique attendue. Enfin, la section 2.4 s'intéresse à la composition des transformations certifiantes.

2.1 Tâches de preuves et transformations en Why3

Le formalisme logique de Why3 [5] est basé sur celui de la logique classique du premier ordre. Les termes sont typés, les types étant possiblement polymorphes et les variables de type étant implicitement quantifiées de façon prénexe sur chaque déclaration globale. Ce formalisme fournit un certain nombre d'extensions syntaxiques et de théories intégrées (entre autres, l'arithmétique des entiers et des réels). Les extensions les plus notables sont les types algébriques et filtrage par motif, les définitions locales (`let tau = 2.0 * pi in ...`) et les expressions conditionnelles (`if c then ... else ...`), celles-ci étant admises aussi bien dans les formules que dans les termes. Why3 fournit des éléments de logique d'ordre supérieur *via* une théorie prédéfinie qui introduit le type « flèche » et le symbole d'application.

Les *tâches de preuve* dans Why3 sont composées d'une liste de prémisses (axiomes et lemmes, déclarations et définitions de types, de fonctions et de prédicats) et d'un but : un énoncé logique à démontrer dans ce contexte. La sémantique d'une tâche est simplement celle d'un séquent logique à un seul but écrit dans la signature correspondante. Les *transformations logiques* sont essentiellement des fonctions OCaml implantées dans le code source de Why3 (ou ajoutées avec un greffon) qui convertissent une tâche T en une liste des tâches $T_1, \ldots, T_n$. Une transformation est *correcte* si la validité des tâches générées implique la validité de la tâche initiale. Ainsi, les constructions étendues avec `let` ou `if` peuvent être éliminées d'une tâche grâce à des transformations logiques appropriées. Cela permet par exemple d'appeler un prouveur qui ne supporte pas ces constructions. De même, c'est grâce à une autre transformation logique que les

constructions d'ordre supérieur peuvent être éliminées, les λ-termes étant extraits des formules et convertis en définitions de symboles logiques de type « flèche ».

L'implantation OCaml de Why3 n'expose pas à l'auteur d'une transformation un accès libre aux structures de termes et de tâches. Au contraire, Why3 fournit une API défensive basée sur des constructeurs et destructeurs malins, qui permettent par exemple de manipuler les lieurs en protégeant contre les problèmes de capture de noms de variables.

2.2 Extraction du contenu logique des tâches

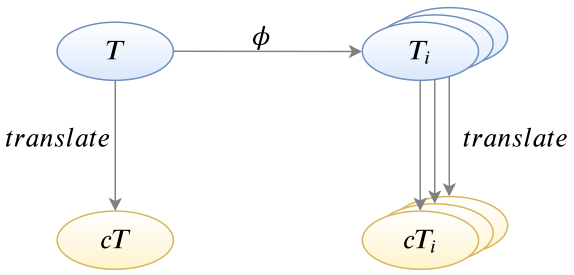
Nous introduisons un nouveau type `ctask` pour représenter les tâches de preuve. Par rapport à la représentation Why3 des tâches, les `ctask` expriment uniquement le contenu logique de celles-ci, sans mentionner les déclarations de types ni les signatures des symboles de fonction. De plus, nous avons choisi de permettre plusieurs buts dans ces `ctask`, comme en calcul des séquents. Les buts et les hypothèses sont alors traités de manière symétrique, nous verrons dans cette section que cela permet de limiter le nombre de certificats différents pour deux raisons. D'une part, un même certificat peut être utilisé pour deux règles duales (voir la FIGURE 2 en section 2.3). D'autre part, les certificats sont alors naturellement plus expressifs. Par exemple la règle d'introduction d'une implication à droite est dérivée à partir celle de l'introduction de la disjonction à droite (voir l'exemple 2.4).

Les `ctask` sont ainsi représentées par des séquents, ce que nous notons de la façon suivante :

$$H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$$

En l'absence de signature explicite, la validité d'une `ctask` exprime implicitement une validité pour toute interprétation possible des symboles qui y apparaissent librement.

Nous définissons une fonction de traduction *translate* qui, suivant l'approche sceptique, sera utilisée à chaque application d'une transformation ϕ, à la fois sur la tâche initiale T et sur les tâches résultantes T_i, comme indiqué sur la figure ci-contre.



Notons que la traduction d'une tâche passe aussi par la traduction des formules qui la composent. On a choisit une représentation *locally nameless* afin de simplifier le traitement des transformations qui effectuent des réécritures dans les formules.

On souhaite certifier que la validité de $T_1, \ldots T_n$ entraîne celle de T, on doit donc vérifier que la validité de $cT_1, \ldots, cT_n$ entraîne celle de cT et faire confiance à la traduction de tâches. Cette dernière apparaît dans les deux sens : la validité de cT doit entraîner celle de la tâche Why3 T et la validité de chaque tâche Why3 T_i doit entraîner celle de la tâche traduite cT_i. Il doit donc y avoir équi-validité entre une tâche et sa traduction. En ce sens, la traduction fait partie de notre base de confiance. Si, par exemple, la traduction donne toujours $\vdash G : \top$, toutes les transformations seront automatiquement vérifiées. Heureusement, la traduction d'une tâche Why3 est un procédé simple qui revient essentiellement à retirer des champs dans un enregistrement.

Dans la suite, nous supposerons qu'une tâche Why3 ne diffère pas de sa traduction d'un point de vue logique. On peut alors, sans ambiguïté, utiliser la notation en séquents aussi bien pour les tâches Why3 que pour les *ctask*.

$$
\begin{array}{rcll}
\mathit{certif} & ::= & \mathsf{Hole} & \mid\ \mathsf{Swap\_neg}(\mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Trivial}(\mathit{ident}) & \mid\ \mathsf{Destruct}(\mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Axiom}(\mathit{ident}, \mathit{ident}) & \mid\ \mathsf{Weakening}(\mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Cut}(\mathit{ident}, \mathit{cterm}, \mathit{certif}, \mathit{certif}) & \mid\ \mathsf{Intro\_quant}(\mathit{ident}, \mathit{ident}, \mathit{certif}) \\
& \mid & \mathsf{Split}(\mathit{ident}, \mathit{certif}, \mathit{certif}) & \mid\ \mathsf{Inst\_quant}(\mathit{ident}, \mathit{ident}, \mathit{cterm}, \mathit{certif}) \\
& \mid & \mathsf{Unfold}(\mathit{ident}, \mathit{certif}) & \mid\ \mathsf{Rewrite}(\mathit{ident}, \mathit{ident}, \mathit{path}, \mathit{bool}, \mathit{certif}^*)
\end{array}
$$

FIGURE 1 – Langage des certificats.

$$
\frac{}{\Gamma \vdash \Delta \overset{\mathtt{Hole}}{\Longleftarrow} \{\Gamma \vdash \Delta\}}
\qquad
\frac{}{\Gamma, H : A \vdash \Delta, G : A \overset{\mathtt{Axiom}(H,G)}{\Longleftarrow} \emptyset}
$$

$$
\frac{\Gamma, P : A \vdash \Delta \overset{c_1}{\Longleftarrow} S_1 \qquad \Gamma, P : B \vdash \Delta \overset{c_2}{\Longleftarrow} S_2}{\Gamma, P : A \vee B \vdash \Delta \overset{\mathsf{Split}(P,c_1,c_2)}{\Longleftarrow} S_1 \cup S_2}
\qquad
\frac{\Gamma \vdash \Delta, P : A \overset{c_1}{\Longleftarrow} S_1 \qquad \Gamma \vdash \Delta, P : B \overset{c_2}{\Longleftarrow} S_2}{\Gamma \vdash \Delta, P : A \wedge B \overset{\mathsf{Split}(P,c_1,c_2)}{\Longleftarrow} S_1 \cup S_2}
$$

$$
\frac{\Gamma, H_1 : A, H_2 : B \vdash \Delta \overset{c}{\Longleftarrow} S}{\Gamma, P : A \wedge B \vdash \Delta \overset{\mathsf{Destruct}(P,H_1,H_2,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma \vdash \Delta, H_1 : A, H_2 : B \overset{c}{\Longleftarrow} S}{\Gamma \vdash \Delta, P : A \vee B \overset{\mathsf{Destruct}(P,H_1,H_2,c)}{\Longleftarrow} S}
$$

$$
\frac{\Gamma, P : Q\ y \vdash \Delta \overset{c}{\Longleftarrow} S \qquad y \text{ fresh}}{\Gamma, P : \exists x.Q\ x \vdash \Delta \overset{\mathsf{Intro\_quant}(P,y,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma \vdash \Delta, P : Q\ y \overset{c}{\Longleftarrow} S \qquad y \text{ fresh}}{\Gamma \vdash \Delta, P : \forall x.Q\ x \overset{\mathsf{Intro\_quant}(P,y,c)}{\Longleftarrow} S}
$$

$$
\frac{\Gamma \vdash \Delta, P : \exists x.Q\ x, G : Q\ t \overset{c}{\Longleftarrow} S}{\Gamma \vdash \Delta, P : \exists x.Q\ x \overset{\mathsf{Inst\_quant}(P,G,t,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma, P : \forall x.Q\ x, H : Q\ t \vdash \Delta \overset{c}{\Longleftarrow} S}{\Gamma, P : \forall x.Q\ x \vdash \Delta \overset{\mathsf{Inst\_quant}(P,H,t,c)}{\Longleftarrow} S}
$$

$$
\frac{\Gamma \vdash \Delta, P : \neg A \vee B \overset{c}{\Longleftarrow} S}{\Gamma \vdash \Delta, P : A \Rightarrow B \overset{\mathsf{Unfold}(P,c)}{\Longleftarrow} S}
\qquad
\frac{\Gamma, P : A \vdash \Delta \overset{c}{\Longleftarrow} S}{\Gamma \vdash \Delta, P : \neg A \overset{\mathsf{Swap\_neg}(P,c)}{\Longleftarrow} S}
$$

FIGURE 2 – Règles de dérivation du jugement $T \overset{c}{\Longleftarrow} S$ (extrait).

2.3 Certificats à trous

On suit l'approche sceptique : les transformations sont instrumentées afin de produire un certificat. La particularité de notre approche vient du fait que les transformations ne closent pas nécessairement la tâche courante et les certificats doivent refléter cette particularité. Un constructeur spécial, `Hole`, indique un « trou » dans un certificat, chaque trou devra correspondre à une tâche produite par cette transformation.

Definition 2.1. *Le langage des certificats est défini par la grammaire donnée figure 1, où le non-terminal cterm représente les termes et le non-terminal ident représente les noms de prémisses, de buts, ou de symboles logiques.*

La sémantique des certificats est définie par un prédicat ternaire $T \overset{c}{\Longleftarrow} S$, reliant une tâche initiale T, un certificat c et un ensemble de tâches S, qui affirme que le certificat c garantit que la validité des tâches de S entraîne celle de T. La figure 2 donne un extrait de la définition inductive de ce prédicat.

Ces règles sont similaires à celles du calcul des séquents, nous pouvons en particulier énoncer la propriété suivante qui se démontre par récurrence sur l'arbre de dérivation de $T \overset{\mathcal{C}}{\Leftarrow} S$.

Proposition 2.2. *Pour tout certificat c, tâche T et ensemble de tâches S, si $T \overset{\mathcal{C}}{\Leftarrow} S$ alors la validité de l'ensemble des tâches de S entraîne la validité de T.*

Voici quelques exemples. Le certificat `Hole` est utilisé afin de garantir la validité des transformations qui ont des tâches résultantes. Celui-ci garantit notamment la validité d'une transformation identité. Le certificat $\mathtt{Axiom}(H, G)$ garantit la validité d'une transformation qui clôt une tâche ayant un but G identique à une prémisse H. Enfin, le certificat $\mathsf{Split}(P, c_1, c_2)$ garantit la validité d'une transformation qui commence par scinder une prémisse ou un but P et dont la validité du reste de la transformation sur ces deux tâches serait assurée par c_1 et c_2. Notons que le même certificat, Split, est utilisé pour deux règles duales.

Exemple 2.3. *Considérons la transformation* `split_all` *qui détruit les conjonctions tant que possible et tente d'éliminer à la volée les sous-tâches triviales. En notant $\Gamma := H_1 : A_1,\ H_3 : A_3$, cette transformation appliquée à $\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$ donne la tâche $\Gamma \vdash G : A_2$ et le certificat certif $:= \mathsf{Split}(G, \mathsf{Axiom}(H_1, G), \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))$ garantit la validité de cette application. En effet, on a la dérivation :*

$$\dfrac{\dfrac{}{\Gamma \vdash G : A_1 \overset{\mathsf{Axiom}(H_1,G)}{\Longleftarrow} \emptyset} \qquad \dfrac{\dfrac{}{\Gamma \vdash G : A_2 \overset{\mathsf{Hole}}{\Longleftarrow} \{\Gamma \vdash G : A_2\}} \qquad \dfrac{}{\Gamma \vdash G : A_3 \overset{\mathsf{Axiom}(H_3,G)}{\Longleftarrow} \emptyset}}{\Gamma \vdash G : A_2 \wedge A_3 \overset{\mathsf{Split}(G,\mathsf{Hole},\mathsf{Axiom}(H_3,G))}{\Longleftarrow} \{\Gamma \vdash G : A_2\}}}{\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3) \overset{certif}{\Longleftarrow} \{\Gamma \vdash G : A_2\}}$$

Exemple 2.4 (Paradoxe du buveur)**.** *Donnons-nous un type A habité par a et posons dr la formule $\exists x.(P\,x \Rightarrow \forall y.P\,y)$. Il existe un certificat c tel que $(\vdash G_1 : dr) \overset{\mathcal{C}}{\Leftarrow} \emptyset$.*

En effet, posons $\mathsf{Intro}(G, P, c) := \mathsf{Unfold}(G, \mathsf{Destruct}(G, P, G, \mathsf{Swap\_neg}(P, c)))$. Pour deux formules A et B données, la dérivation du schéma de gauche suivant est abrégée par celle du schéma de droite (notation avec double barre) :

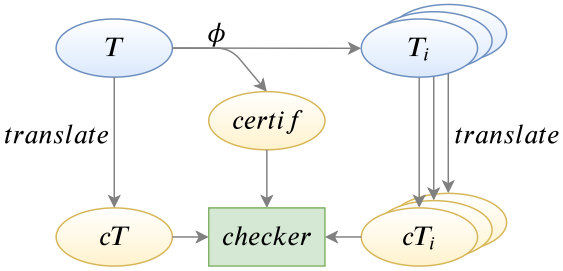
$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, P : A \vdash \Delta, G : B \overset{\mathcal{C}}{\Leftarrow} S}{\Gamma \vdash \Delta, P : \neg A, G : B \overset{\mathsf{Swap\_neg}(P,c)}{\Longleftarrow} S}}{\Gamma \vdash \Delta, G : \neg A \vee B \overset{\mathsf{Destruct}(G,P,G,\ldots)}{\Longleftarrow} S}}{\Gamma \vdash \Delta, G : A \Rightarrow B \overset{\mathsf{Unfold}(G,\ldots)}{\Longleftarrow} S}}{\qquad}$$

$$\dfrac{\Gamma, P : A \vdash \Delta, G : B \overset{\mathcal{C}}{\Leftarrow} S}{\Gamma \vdash \Delta, G : A \Rightarrow B \overset{\mathsf{Intro}(G,P,c)}{\Longleftarrow} S}$$

où les ... abrègent le certificat du jugement situé au-dessus. On a alors la dérivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{H_1 : P\,a, H_2 : P\,y \vdash G_1 : dr, G_2 : P\,y, G_3 : \forall y.P\,y \overset{\mathsf{Axiom}(H_2,G_2)}{\Longleftarrow} \emptyset}}{H_1 : P\,a \vdash G_1 : dr, G_2 : P\,y, G_3 : P\,y \Rightarrow \forall y.P\,y \overset{\mathsf{Intro}(G_3,H_2,\ldots)}{\Longleftarrow} \emptyset}}{H_1 : P\,a \vdash G_1 : dr, G_2 : P\,y \overset{\mathsf{Inst\_quant}(G_1,G_3,y,\ldots)}{\Longleftarrow} \emptyset}}{H_1 : P\,a \vdash G_1 : dr, G_2 : \forall y.P\,y \overset{\mathsf{Intro\_quant}(G_2,y,\ldots)}{\Longleftarrow} \emptyset}}{\vdash G_1 : dr, G_2 : P\,a \Rightarrow \forall y.P\,y \overset{\mathsf{Intro}(G_2,H_1,\ldots)}{\Longleftarrow} \emptyset}}{\vdash G_1 : dr \overset{\mathsf{Inst\_quant}(G_1,G_2,a,\ldots)}{\Longleftarrow} \emptyset}$$

Vérificateurs de certificats. Un vérificateur de certificats est une procédure qui, étant donnés une tâche T, un certificat c et un ensemble de tâches S, tente de vérifier que $T \overset{c}{\Leftarrow} S$ est vrai. En pratique, un tel vérificateur *checker* s'applique à une tâche traduite cT, au certificat produit *certif* et à l'ensemble des tâches résultantes traduites cT_i, comme indiqué sur la figure adjacente.



Definition 2.5. *Un tel vérificateur est* correct *si, lorsqu'il répond positivement, alors* $T \overset{c}{\Leftarrow} S$ *est dérivable.*

2.4 Composition des transformations certifiantes

Notre définition de certificats à trous permet une certification *modulaire* selon deux sens : d'une part il est possible de certifier l'action d'une transformation sans devoir nécessairement certifier le traitement ultérieur des sous-tâches résultantes, d'autre part il est possible de composer les certificats pour certifier une transformation qui est obtenue par composition de transformations déjà certifiantes. Nous supposerons ici que la composition de deux transformations ϕ_1 et ϕ_2 donne la transformation qui applique d'abord ϕ_1 à la tâche initiale et ensuite ϕ_2 à toutes les tâches résultant de l'application de ϕ_1.

Pour obtenir le certificat de l'application de ϕ_1 suivie de ϕ_2, ϕ_1 et ϕ_2 étant des transformations certifiantes, on procède comme suit : (1) on applique ϕ_1 à la tâche initiale, on obtient un certificat c et une liste de tâches lt, où chaque Hole de c correspond exactement à une tâche de lt ; (2) on remplace chacun de ces Hole de c par le certificat renvoyé par l'application de ϕ_2 sur la tâche correspondant à ce Hole. Le certificat c avec ces remplacements est le certificat recherché.

Exemple 2.6 (Suite de l'exemple 2.3)**.** *La transformation* `split_all` *peut être vue comme l'itération d'une transformation* ϕ *qui clôt la tâche initiale si elle est triviale et qui, sinon, décompose au plus une fois une conjonction dans un but. L'application de* ϕ *sur* $\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$ *donne*

$$[\Gamma \vdash G : A_1; \Gamma \vdash G : A_2 \wedge A_3],\ \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Hole})$$

On peut réappliquer ϕ *sur chacune des deux tâches résultantes :*

$$\phi(\Gamma \vdash G : A_1) = [],\ \mathsf{Axiom}(H_1, G)$$
$$\phi(\Gamma \vdash G : A_2 \wedge A_3) = [\Gamma \vdash G : A_2; \Gamma \vdash G : A_3],\ \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Hole})$$

À partir de $\phi(\Gamma \vdash G : A_2) = [\Gamma \vdash G : A_2],\ \mathsf{Hole}$ *et de* $\phi(\Gamma \vdash G : A_3) = [],\ \mathsf{Axiom}(H_3, G)$*, on retrouve le certificat pour l'itération de* ϕ *en composant les certificats comme indiqué ci-dessus :*

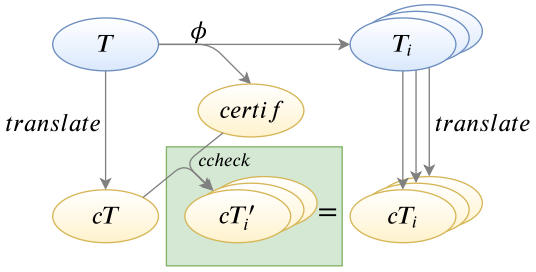
$$\mathsf{Split}(G, \mathsf{Axiom}(H_1, G), \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))$$

3 Vérificateur codé en OCaml

Le premier vérificateur que nous proposons est implanté en OCaml. Dans cette première approche, nous définissons une version exécutable du jugement $T \overset{c}{\Leftarrow} S$ sous la forme d'une

fonction OCaml ccheck qui, à partir d'une tâche T et d'un certificat c, reconstruit une *liste* de tâches L dont les éléments sont ceux de S.

Une transformation certifiante produit un certificat, ce qui permet de vérifier son résultat. Dans le cas du vérificateur en OCaml cette vérification se fait en appelant la fonction ccheck selon le schéma ci-après.



À partir de T, une tâche Why3, la transformation ϕ considérée donne *certif* et une liste de tâches T_i. La fonction *translate* calcule alors les versions "abstraites" cT et cT_i des tâches de départ et d'arrivée, on appelle ccheck sur *certif* et cT pour obtenir une liste cT'_i et on vérifie point à point que les listes cT_i et cT'_i sont identiques.

Notons que dans cette implantation, nous avons délibérément choisi de produire une liste de tâches et de ne pas chercher à autoriser les permutations dans cette liste. Ce vérificateur est donc en ce sens incomplet (il ne « décide » pas tous les $T \overset{certif}{\Longleftarrow} S$) mais a le mérite d'être calculable efficacement. En pratique, cette restriction n'est pas prohibitive car il est toujours possible de modifier le certificat afin de faire correspondre ses « trous » avec la liste de tâches renvoyée.

La base de confiance de cette approche est constituée de la fonction d'abstraction des tâches *translate* et de la fonction ccheck.

Réalisation de la fonction ccheck L'implantation de ccheck procède naturellement par récurrence sur le certificat, et suivant les règles qui définissent le jugement $T \overset{c}{\Leftarrow} S$. Ce calcul vérifie les conditions d'application des règles, et lève des exceptions quand ce n'est pas le cas. Détaillons le fragment de code correspondant au constructeur $\mathsf{Split}(P, c_1, c_2)$: ccheck vérifie que la formule donnée par l'identifiant P est scindable, c'est-à-dire que c'est une conjonction si c'est un but ou une disjonction si c'est une prémisse. Si c'est le cas, ccheck s'appelle récursivement, par exemple dans le cas d'un but :

$$\mathsf{ccheck}\ (\mathsf{Split}(P, c_1, c_2))\ (\Gamma \vdash \Delta, P : A_1 \wedge A_2) \equiv$$
$$\mathsf{ccheck}\ c_1\ (\Gamma \vdash \Delta, P : A_1)\ @\ \mathsf{ccheck}\ c_2\ (\Gamma \vdash \Delta, P : A_2)$$

où @ est le symbole infixe pour la concaténation de listes.

Exemple 3.1 (Suite de l'exemple 2.3)**.** *Pour vérifier que l'application de* `split_all` *à la tâche* $\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$ *est bien certifiée par* $c := \mathsf{Split}(G, \mathsf{Axiom}(H_1, G), \mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))$ *on calcule :*

$$\begin{aligned}
&\mathit{ccheck}\ c\ (\Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)) \\
&\quad = \mathit{ccheck}\ (\mathsf{Axiom}(H_1, G))\ (\Gamma \vdash G : A_1)\ @ \\
&\qquad \mathit{ccheck}\ (\mathsf{Split}(G, \mathsf{Hole}, \mathsf{Axiom}(H_3, G)))\ (\Gamma \vdash G : A_2 \wedge A_3) \\
&\quad = [] \ @\ \mathit{ccheck}\ \mathsf{Hole}\ (\Gamma \vdash G : A_2)\ @\ \mathit{ccheck}\ (\mathsf{Axiom}(H_3, G))\ (\Gamma \vdash G : A_3) \\
&\quad = [\Gamma \vdash G : A_2]
\end{aligned}$$

qui est bien la tâche résultante attendue.

Proposition 3.2 (Correction de ccheck)**.** *Pour tout certificat c et tâche T, si* ccheck c T *renvoie une liste L, alors il existe une dérivation de $T \overset{\mathcal{L}}{\Leftarrow} S$, où S est l'ensemble des éléments de L.*

La preuve de cette proposition peut se faire par récurrence sur le certificat, en prenant tous les cas un à un. Il s'agit d'une preuve que l'on pourrait faire avec un assistant de preuve. Une telle preuve peut vite devenir fastidieuse en augmentant le nombre de règles de certificats. Nous nous sommes donc plutôt concentrés sur une deuxième approche, présentée ci-après.

4 Vérificateur basé sur Dedukti

Notre deuxième approche pour réaliser un vérificateur vise à se passer d'une preuve formalisée d'un théorème de correction comme la proposition 3.2. Cette autre approche utilise le vérificateur de preuve universel Dedukti. L'intérêt est d'utiliser son mécanisme de règles de réécriture pour encoder simplement nos certificats. À chaque fois qu'une transformation Why3 est utilisée, une preuve Dedukti de correction est produite et peut être vérifiée par une implantation d'un vérificateur de types pour ce langage [1].

Plus précisément, on propose un encodage des tâches de preuves dans Dedukti, sous la forme d'un plongement superficiel : une tâche T est encodée en une formule Dedukti $\hat{T}$. Si sur une tâche T on applique une transformation certifiante produisant une liste de tâches T_i et un certificat c, alors nous construisons un terme Dedukti qui est candidat pour une preuve de $\hat{T_1}$`->`$\cdots$`->`$\hat{T_n}$`->`$\hat{T}$, où `->` est la flèche de Dedukti. Ce terme candidat est naturellement destiné à être transmis à Dedukti pour vérification.

Dans un premier temps nous rappelons un encodage existant de la logique du premier ordre en Dedukti, puis nous présentons notre plongement superficiel.

Dans cette deuxième approche, la base de confiance est constituée de la fonction d'abstraction des tâches Why3, de la fonction de traduction des tâches abstraites vers Dedukti, et bien sûr de Dedukti lui-même et de l'encodage de la logique du premier ordre utilisé. En revanche, la fonction de traduction d'un certificat en un terme de preuve pour Dedukti n'est pas dans la base de confiance. En effet, si l'étape de traduction contient un *bug* alors la vérification risque d'échouer, mais, si elle réussit, la transformation Why3 initiale a bien fonctionné correctement sur la tâche considérée.

4.1 Logique du premier ordre en Dedukti

Dedukti [3] est un vérificateur de types pour le $\lambda\Pi$-calcul modulo, un formalisme logique basé sur un $\lambda\Pi$-calcul extensible avec des règles de réécriture. Il possède trois constructions : l'abstraction $\lambda x : A.u$, notée [2] `(x:A) => u` ; le produit dépendant $\Pi x : A.B$, noté `(x:A) -> B` ; et la possibilité de définir des règles de réécriture qui s'ajoutent à la conversion usuelle du $\lambda\Pi$-calcul, notées `[vars] l --> r`, signifiant « `l` se réécrit en `r` », où `[vars]` liste les variables libres apparaissant dans `l` et `r`.

La possibilité d'ajouter des règles de calcul arbitraires permet en toute généralité de construire des contextes incohérents, de la même façon que l'on peut ajouter n'importe quel axiome en Coq. Mais des encodages de la plupart des systèmes de preuve dans ce formalisme ont été étudiés et montrés corrects. Notre vérificateur se base actuellement sur un encodage de

1. À notre connaissance, deux implantations sont disponibles, `dkcheck` et `lambdapi`. Nous utilisons actuellement le vérificateur `dkcheck` pour des raisons techniques (voir https://github.com/Deducteam/lambdapi/issues/243).
2. Dans la suite, le type `A` sera omis lorsque l'on peut le déduire aisément du contexte.

la logique du premier ordre intuitioniste dans ce système, défini dans la bibliothèque de Dedukti et que nous rappelons ci-dessous.

Deux types sont introduits pour représenter les termes et les propositions de la logique du premier ordre, puis le type `Prop` est construit par les connecteurs logiques :

```
Term : Type.
Prop : Type.
true   : Prop.
false  : Prop.
not    : Prop -> Prop.
and    : Prop -> Prop -> Prop.
or     : Prop -> Prop -> Prop.
imp    : Prop -> Prop -> Prop.
forall : (Term -> Prop) -> Prop.
exists : (Term -> Prop) -> Prop.
```

et le type `Term` par la signature de la théorie considérée (par exemple, les entiers). On notera que l'encodage des quantificateurs utilise une syntaxe abstraite d'ordre supérieur.

Le point clé de l'encodage est un plongement `prf` des preuves de la logique du premier ordre dans la logique sous-jacente de Dedukti, qui donne leur sémantique aux constantes à travers des règles de réécriture (et une preuve directe dans le cas de `true`) :

```
prf : Prop -> Type.
tt:   prf true.
[]    prf false       --> (C:Prop) -> prf C
[A]   prf (not A)     --> prf A -> prf false
[A,B] prf (and A B)   --> (C:Prop) -> (prf A -> prf B -> prf C) -> prf C
[A,B] prf (or  A B)   --> (C:Prop) -> (prf A -> prf C)
                                   -> (prf B -> prf C) -> prf C
[A,B] prf (imp A B)   --> prf A -> prf B
[P]   prf (forall P)  --> (x:Term) -> prf (P x)
[P]   prf (exists P)  --> (A:Prop) -> ((x:Term) -> prf (P x) -> prf A)
                                   -> prf A
tnd: (A:Prop) -> prf (or A (not A))
```

Ainsi, la constante `false` est plongée dans un terme de $\lambda\Pi$ encodant la contradiction (`(C:Prop) -> prf C`, indiquant que toute proposition est vraie), et ainsi de suite. Notons également que comme nous travaillons en logique classique, nous ajoutons l'axiome `tnd`

Exemple 4.1. *Si on fixe A de type* `Prop` *et P de type* `Term -> Prop` *alors donner une preuve de $\forall x.\,(A \wedge (\forall y.\,P\,y)) \Rightarrow ((P\,x) \wedge A)$ consiste à donner un terme de type*

```
prf (forall ((x:Term) =>
     imp (and A (forall ((y : Term) => P y)) (and (P x) A))).
```

Par les règles ci-dessus et β-réduction, ce terme est convertible à

```
(x:Term) ->
 ((Q:Prop) -> (prf A -> ((y:Term) -> prf (P y)) -> prf Q) -> prf Q) ->
  ((Q:Prop) -> (prf (P x) -> prf A -> prf Q) -> prf Q)
```

et une preuve en est donc

```
x => hAP => Q => hQ => hAP Q (hA => hP => hQ (hP x) hA).
```

Notre *back-end* Dedukti traduit les tâches comme des types Dedukti qui s'appuient sur l'encodage ci-dessus. Le certificat est lui-même traduit vers un terme Dedukti, et la vérification consiste à s'assurer que ce terme a bien ce type, selon le principe de l'isomorphisme de Curry-Howard.

4.2 Traduction des tâches

Une tâche T est encodée en un type Dedukti $\hat{T}$ à l'aide de nouvelles constructions :

```
empty: Prop
hyp: Prop -> Prop -> Prop
goal: Prop -> Prop -> Prop
[] empty --> false
[H,S] hyp H S --> imp H S
[G,S] goal G S --> imp (not G) S
```

Si $T = H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$ alors $\hat{T}$ est

$$\texttt{prf}\ (\texttt{hyp}\ \ A_1\ \ (\texttt{hyp}\ \ A_2\ \ (\cdots\ \ (\texttt{hyp}\ \ A_m\ \ (\texttt{goal}\ \ B_1 (\cdots\ \ (\texttt{goal}\ \ B_n\ \ \texttt{empty}) \cdots))) \cdots)))$$

ce qui correspond à la formule $A_1 \Rightarrow ... \Rightarrow A_m \Rightarrow \neg B_1 \Rightarrow ... \Rightarrow \neg B_n \Rightarrow \bot$, notation que nous utiliserons dans la suite pour l'encodage des formules et des tâches Why3 en Dedukti.

Il faut remarquer que nous suivons l'approche usuelle d'un plongement superficiel, en utilisant l'implication de Dedukti. Une approche plus naïve serait de traduire la tâche $H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$ en $(A_1 \wedge ... \wedge A_m) \Rightarrow (B_1 \vee ... \vee B_n)$ et de procéder de manière similaire pour l'implication entre la conjonction des tâches résultantes et la tâche initiale. Cette approche naïve demande un traitement explicite du contexte qui devient rapidement très lourd et n'est donc en pratique pas utilisable.

Exemple 4.2. *Considérons la tâche $H_2 : B \vee C \vdash$ et une transformation ϕ qui scinde H_2. ϕ produit les sous-tâches $H_2 : B \vdash$ et $H_2 : C \vdash$. Le type Dedukti qui correspondrait à la traduction naïve serait $(B \Rightarrow \bot) \wedge (C \Rightarrow \bot) \Rightarrow B \vee C \Rightarrow \bot$. Supposons maintenant que l'on ait obtenu un terme* `t` *qui a ce type et considérons le nouveau problème où l'on ajoute l'hypothèse A. On applique ϕ à la tâche $H_1 : A, H_2 : B \vee C \vdash$ et on obtient les tâches $H_1 : A, H_2 : B \vdash$ et $H_1 : A, H_2 : C \vdash$. Il n'est pas facile d'obtenir, à partir de* `t`*, un terme de type $(A \wedge B \Rightarrow \bot) \wedge (A \wedge C \Rightarrow \bot) \Rightarrow A \wedge (B \vee C) \Rightarrow \bot$, car il faudrait construire et déconstruire plusieurs fois des conjonctions.*

Avec notre plongement superficiel, si on a construit le terme `t` *de type $(B \Rightarrow \bot) \Rightarrow (C \Rightarrow \bot) \Rightarrow B \vee C \Rightarrow \bot$ et que l'on cherche à obtenir un terme de type $(A \Rightarrow B \Rightarrow \bot) \Rightarrow (A \Rightarrow C \Rightarrow \bot) \Rightarrow A \Rightarrow B \vee C \Rightarrow \bot$, il suffit d'écrire :* `cT1 => cT2 => a => bc => t (cT1 a) (cT2 a) bc`.

4.3 Traduction et élaboration du certificat

4.3.1 Définition de termes de preuve

Le terme de preuve dont on vérifiera qu'il a le type défini ci-dessus est obtenu à partir du certificat. Pour ce faire, dans un premier temps, nous associons un terme Dedukti à chacune des règles d'inférence de $T \overset{\mathcal{L}}{\Leftarrow} S$. Le type de ce terme est obtenu en ignorant le contexte et les tâches résultantes de cette règle et en prenant sa formule logique équivalente.

Illustrons cette démarche sur la règle qui scinde une disjonction dans une prémisse :

$$\frac{\Gamma, P : A \vdash \Delta \overset{c_1}{\Leftarrow} S_1 \qquad \Gamma, P : B \vdash \Delta \overset{c_2}{\Leftarrow} S_2}{\Gamma, P : A \vee B \vdash \Delta \overset{\mathsf{Split}(P, c_1, c_2)}{\Longleftarrow} S_1 \cup S_2}$$

En lui retirant le contexte Γ et Δ, le certificat c et les tâches résultantes S, cette règle devient :

$$\frac{P : A \vdash \qquad P : B \vdash}{P : A \vee B \vdash}$$

On définit donc une fois pour toutes en Dedukti le terme suivant

```
def split_hyp (A : Prop) (B : Prop) :
    prf (hyp A empty) -> prf (hyp B empty) ->
    prf (hyp (or A B) empty)
:= cT1 => cT2 => H => or_elim A B H false cT1 cT2.
```

À titre d'exemple, voici également le terme Dedukti introduit pour le cas d'une conjonction dans une prémisse :

```
def destruct_hyp (A : Prop) (B : Prop) :
    prf (hyp A (hyp B empty)) ->
    prf (hyp (and A B) empty)
:= cT1 => H => cT1 (and_elim_1 A B H) (and_elim_2 A B H).
```

4.3.2 Traduction du certificat

Remarquons que l'on souhaite obtenir un terme de preuve d'un type fourni par la traduction des tâches et qui a donc la forme suivante :

$$cT_1 \Rightarrow ... \Rightarrow cT_n \Rightarrow T$$

où les cT_i sont les tâches résultantes et T est la tâche initiale qui est elle-même de la forme $H_1 \Rightarrow ... \Rightarrow H_m \Rightarrow \bot$.

La principale difficulté pour obtenir ce terme de preuve vient de la gestion des noms. Premièrement, il faut générer et se souvenir de noms pour les tâches résultantes (le terme de preuve commence par `cT1 => ... => cTn =>`). La suite du terme de preuve est obtenue en introduisant les prémisses de la tâche initiale (`H1 => ... => Hm =>`). Enfin, le cœur du terme de preuve provient de la traduction du certificat, ce dernier indiquant les noms des hypothèses intermédiaires. La traduction d'un constructeur de certificat utilise le terme de preuve correspondant (ceux-ci étant définis en 4.3.1), à l'exception du constructeur Hole qui est traduit par la tâche correspondante appliquée aux identificateurs de ses prémisses.

Exemple 4.3. *Supposons qu'une application d'une certaine transformation sur la tâche* $H : A \vee (B \wedge C) \vdash$ *donne les tâches* $H : A \vdash$ *et* $H_b : B, H_c : C \vdash$ *et le certificat* Split (*H*, Hole, Destruct (*H*, H_b, H_c, Hole)). *On introduit les tâches résultantes (*`cT1 => cT2 => ...`*), la prémisse de la tâche initiale (*`H => ...`*) puis on suit le certificat qui fait d'abord un* Split *sur la prémisse H (*`split_hyp (H => c1) (H => c2) H`*) où* `c1` *suit le sous certificat gauche de* Split. *Celui-ci est le premier* Hole *rencontré et est donc remplacé par la première tâche appliquée aux ident de ses prémisses, c'est-à-dire par* `cT1 H`. *Pour* `c2`, *on suit le certificat* Destruct*(H, H_b, H_c, Hole), ce qui nous donne* `destruct_hyp (Hb => Hc => c3) H`. *Cette fois-ci* `c3` *suit le deuxième* Hole *rencontré et est donc remplacé par la deuxième tâche appliquée aux ident de ses prémisses, soit* `cT2 Hb Hc`. *En résumé, on vérifie avec Dedukti que le terme*

```
cT1 => cT2 => H =>
 split_hyp (H => cT1 H)
           (H => destruct_hyp (Hb => Hc => cT2 Hb Hc) H) H
```

a bien le type $(A \Rightarrow \bot) \Rightarrow (B \Rightarrow C \Rightarrow \bot) \Rightarrow A \vee (B \wedge C) \Rightarrow \bot$.

4.3.3 Élaboration du certificat

Le type de nos certificats est relativement peu verbeux. En particulier, les constructeurs ne précisent pas quelles formules sont manipulées ni si ce sont des prémisses ou des buts. Cette ambiguïté demande un effort supplémentaire lors de la traduction des certificats vers des termes preuve Dedukti : c'est ce que l'on appelle l'élaboration des certificats. Les certificats sont reparcourus en partant de la tâche initiale, on peut alors en déduire le contexte d'application d'une étape de certificat. Les types des termes de preuve correspondant aux règles sont généralisés en les variables libres qui y apparaissent. Du point de vue de la génération du terme de preuve, il s'agit de leur donner des paramètres qui explicitent ces formules.

Exemple 4.4. *Dans l'exemple précédent,* Split *a été traduit en* `split_hyp` *car la formule manipulée* `H` *est une prémisse et non un but.* `split_hyp` *prend* A *et* $B \wedge C$ *comme premiers arguments car* `H` *est la disjonction de* A *et de* $B \wedge C$*. De même, les deux premiers aguments de* `destruct_hyp` *sont* B *et* C *car son étape de certificat* Destruct *correspondante manipule l'hypothèse* $B \wedge C$.

5 Expérimentations

Des transformations de Why3, de l'ordre d'une quinzaine, ont été instrumentées pour générer des certificats et les vérifier à la volée lors de chaque utilisation. De nouvelles transformations ont été implantées, la plus complexe d'entre elles étant une transformation nommée `blast`. Elle est obtenue par composition de transformations certifiantes plus élémentaires qui s'appuient uniquement sur la logique du premier ordre : décomposition de conjonctions, de disjonctions, d'implications et d'équivalences, résolution de tâches triviales, introduction d'hypothèses, etc. Notre méthode de composition de transformations certifiantes décrite dans la section 2.4 nous a permis de définir `blast` de façon à ce qu'elle s'appelle naturellement de manière récursive sur les sous-tâches qu'elle génère. Par ailleurs, une version légèrement simplifiée de la transformation `rewrite` permettant de réécrire dans les termes a été rendue certifiante. Cette transformation implante un certain nombre de fonctionnalités de la transformation initiale mais ne supporte pas encore les quantificateurs universels en tête de la formule à réécrire. Le vérificateur OCaml a été inclus dans Why3 et est disponible pour toutes les transformations déjà certifiantes. Il en est de même pour le mécanisme de vérification via Dedukti à l'exception de la transformation `rewrite` (à ce jour). À l'exception de cette dernière, nos expérimentations ont permis de montrer que les certificats générés par nos transformations certifiantes sont validés par Dedukti.

Pour évaluer l'efficacité des implantations de nos vérificateurs, nous avons testé la transformation `blast` sur une famille classique de problèmes pour évaluer les solveurs propositionnels : le principe des tiroirs de Dirichlet. Pour n donné, on ne peut pas mettre n chaussettes dans $n-1$ tiroirs sans qu'un tiroir contienne au moins une paire de chaussettes. La table suivante indique, pour un n donné : le nombre de variables booléennes, et le nombre total d'occurrences des variables dans le problème ; le temps que prend la transformation pour clore le problème et construire le certificat ; la taille de ce certificat (s'il est écrit dans un fichier) ; le temps de la vérification de ce certificat par le vérificateur OCaml ; le temps pris par l'élaboration du fichier Dedukti ; la taille de ce fichier ; et enfin le temps de vérification par Dedukti (appel de `dkcheck`).

nombre de chaussettes	3	4	5
nombre de variables	6	12	20
nombre d'occurrences de variables	18	48	100
temps d'exécution de la transformation (sec)	7×10^{-3}	12	-
taille du certificat (octet)	37×10^{3}	26×10^{6}	-
temps vérification OCaml (sec)	63×10^{-6}	55×10^{-3}	-
taille certificat Dedukti (octet)	89×10^{3}	46×10^{6}	-
temps élaboration Dedukti (sec)	8×10^{-3}	1,9	-
temps vérification Dedukti (sec)	24×10^{-3}	20	-

Le test sur cinq chaussettes ne se termine pas en un temps raisonnable, ce qui n'est pas surprenant car cette famille de problèmes nécessite des preuves sans coupures de taille exponentielle. Pour faire mieux, il faudrait que notre transformation `blast` recherche des preuves avec coupures. On remarque que le vérificateur OCaml est bien plus rapide que Dedukti. En effet, le vérificateur en OCaml utilise une approche réflexive, c'est-à-dire que la vérification est effectuée par un calcul dont les entrées sont les tâches et le certificat. Cela a l'avantage d'être très efficace dans le cas où le moteur de calcul est performant. *A contrario*, le vérificateur en Dedukti repose sur un plongement superficiel qui réduit la vérification à une question de typage dans un cadre où il faut appliquer des réécritures sur les types. On note aussi que la taille du fichier Dedukti est comparable à la taille du certificat. S'il est vrai que l'encodage des termes en Dedukti est linéaire, l'élaboration aurait pu accroître la taille des certificats de façon non-linéaire, ce que l'on n'observe pas ici, probablement parce la plupart des formules sur lesquelles on travaille sont petites.

6 Conclusions, travaux connexes et perspectives

Nous avons présenté un cadre pour valider des transformations logiques « à petit pas » et composables. Cette validation se base sur une approche sceptique : génération d'un certificat pouvant être vérifié *a posteriori* par un outil externe. Notre travail se base sur une notion de certificat à trous, qui s'inspire naturellement fortement de notions analogues dans le contexte de termes-preuve comme les métavariables ou les variables existentielles. Dans le contexte de l'approche sceptique, l'utilisation de certificats à trous est nouvelle à notre connaissance. Elle nous permet de refléter, au sein des certificats, la modularité et la possibilité de chaîner les transformations : (i) les certificats eux-mêmes sont chaînables, grâce à la possibilité de laisser des trous qui seront comblés par la suite ; (ii) la vérification de ces certificats à trous nécessite de comparer les buts ouverts des certificats avec ceux produits par la transformation elle-même. Outre la flexibilité dans l'écriture des transformations apportée par l'approche sceptique, cela permet une très grande modularité puisque les transformations peuvent être instrumentées de manière totalement indépendante.

Cette approche a été implantée dans l'outil de preuve de programmes Why3 pour un ensemble de transformations en logique du premier ordre, afin de valider la démarche. Deux vérificateurs de certificats sont proposés, un non certifié développé en OCaml avec une approche calculatoire, et un développé dans le *framework* Dedukti à l'aide d'un plongement superficiel.

Travaux connexes. De nombreux outils pour la vérification déductive de programmes ont été développés, basés sur des assistants de preuve ou indépendants.

Dans le premier cas, la preuve de programmes nécessite la définition préalable de bibliothèques pour définir un langage de programmation particulier et une logique de programme

correspondante. Il s'agit par exemple de la bibliothèque Iris [14] au-dessus de Coq qui permet de raisonner sur des programmes impératifs avec une logique de séparation ; ou encore de la bibliothèque AutoCorres [12, 11] au-dessus d'Isabelle permettant de raisonner sur des programmes C. Dans un tel contexte, la preuve d'une propriété d'un programme se fait dans l'environnement de preuve assistée sous-jacent. La correction repose sur une sémantique formelle du langage et des preuves des règles de déduction établies une fois pour toutes, ce qui représente un effort de preuve très coûteux et est donc difficile à faire évoluer.

Dans le deuxième cas, les outils s'appuient sur un langage de programmation particulier équipé d'un langage de spécification associé, implantent un générateur d'obligations de preuve, et utilisent des prouveurs automatiques externes tels que les prouveurs SMT. Des exemples de tels environnements sont donnés par Why3, Dafny, Viper, et également des outils pour des langages *mainstream* comme Frama-C pour le C et SPARK pour Ada. Bien que reposant sur des bases théoriques solides, l'implantation des outils, elle-même, et certains aspects pratiques comme l'utilisation de prouveurs automatiques, n'ont pas été vérifiés mécaniquement et peuvent être sources de bugs. Une exception est l'outil F* [15], dont l'encodage de la logique vers SMT a été en partie établi en Coq [1], de manière globale.

L'originalité de notre approche est premièrement de procéder de manière modulaire, en considérant indépendemment des petites transformations, et deuxièmement d'utiliser l'approche sceptique. Cela offre une plus grande maniabilité, en permettant à la fois l'ajout de nouvelles transformations et l'évolution des transformations existantes à faible coût. Pour cela, nous avons proposé un cadre permettant de mêler transformations certifiantes et non certifiantes en ayant confiance dans les transformations certifiantes, grâce à la vérification de leur certificat combiné au contrôle des sous-buts générés, ce qui est à notre connaissance original.

Concernant la vérification des certificats elle-même, nous nous sommes basés sur deux approches classiques : l'approche réflexive, en OCaml, et un plongement superficiel dans un outil de preuve, en Dedukti. L'approche réflexive est connue pour être très efficace, et peut être utilisée de manière non certifiée ou certifiée dans des assistants de preuve intégrant une notion de calcul [13, 2]. Le plongement dans un outil de preuve est connu pour être moins efficace mais sa correction repose sur celle de l'outil de preuve et est donc beaucoup moins coûteuse [9, 8].

Perspectives. Un objectif à court terme est de poursuivre l'application à Why3, afin de rendre certifiantes la plupart de ses transformations les plus couramment utilisées et ainsi d'augmenter la confiance dans cet outil. Cela nécessite d'étendre (a) le type `ctask` des tâches utilisé pour la validation, (b) le format de certificat et (c) nos vérificateurs. La certification des transformations d'élimination des types algébriques et des types polymorphes seront des défis importants. Nous envisageons notamment de nous placer dans une logique d'ordre supérieur.

Afin d'avoir une pleine confiance dans l'outil Why3, un objectif à plus long terme est de certifier de bout en bout la chaîne de Why3 : en aval, lier ce travail avec la vérification des preuves effectuées par les prouveurs automatiques, déjà proposée dans d'autres travaux [2, 7], et en amont, proposer une méthode de certification de la génération d'obligations de preuve. Ce dernier point nécessite de formaliser la sémantique du langage WhyML.

Un passage à l'échelle va nécessairement poser des questions d'efficacité. D'un point de vue programmation, nous souhaitons nous rapprocher de l'implantation concrète des tâches, notamment en étudiant la possibilité de conserver la mémoïsation (par une approche fonctionnelle). D'un point de vue plus théorique, la démarche présentée ici génère des certificats à petits grains et pouvant donc rapidement devenir volumineux lorsque plusieurs étapes sont combinées. Nous souhaitons étudier la compression de ces certificats, et notamment comment elle pourrait être menée à la volée lors de la combinaison de certificats.

Enfin, notre méthode n'est pas propre à Why3 et peut s'appliquer à la certification d'encodages logiques en toute généralité. Nous souhaitons l'utiliser notamment pour certifier « à petits pas » des encodages de logiques d'assistants de preuve vers des prouveurs automatiques, afin de pouvoir combiner ces deux types de prouveurs.

Remerciements. Nous remercions les évaluateurs, ainsi que la présidente du comité de programme Zaynah Dargaye, pour leurs nombreux commentaires et suggestions durant la phase d'accompagnement pastoral.

Références

[1] A. Aguirre. Towards a provably correct encoding from F* to SMT. Master's thesis, Université Paris 7, 2016.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Thery, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[3] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti : a logical framework based on the λΠ-calculus modulo theory, 2016. Available at http://www.lsv.fr/~dowek/Publi/expressing.pdf.

[4] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *22nd International Conference on Types for Proofs and Programs*, 2016.

[5] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *International Workshop on Intermediate Verification Languages*, pages 53–64, 2011. https://hal.inria.fr/hal-00790310.

[6] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[7] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[8] R. Cauderlier and P. Halmagrand. Checking Zenon modulo proofs in Dedukti. In *Proof eXchange for Theorem Proving*, volume 186 of *EPTCS*, pages 57–73, 2015.

[9] E. Contejean. Coccinelle, a Coq library for rewriting. In *Types*, 2008.

[10] S. Dailler, C. Marché, and Y. Moy. Lightweight interactive proving inside an automatic program verifier. In *Formal Integrated Development Environments*, 2018.

[11] D. Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, CSE, UNSW, 2015.

[12] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff : Formal verification of C code without the pain. In *Programming Language Design and Implementation*, pages 429–439. ACM, 2014.

[13] B. Grégoire, L. Théry, and B. Werner. A computational approach to Pocklington certificates in type theory. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2006.

[14] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Programming Languages and Systems*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer, 2017.

[15] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages*, pages 256–270. ACM, 2016.

Expérimentations pédagogiques en Learn-OCaml

Outils et méthodologie pour les traits avancés du langage

Loïc Sylvestre[1] Emmanuel Chailloux[2]

[1] Sorbonne Université, F-75005 Paris, France
loic.sylvestre@etu.sorbonne-universite.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
emmanuel.chailloux@lip6.fr

Résumé

Learn-OCaml est un logiciel libre destiné à l'enseignement du langage OCaml : il offre un environnement d'exercices de programmation à correction automatisée. On présente dans cet article un support pour la séparation et la réutilisation du code de correction. Cette extension donne lieu, dans un second temps, à l'élaboration d'une bibliothèque de test visant à faciliter l'écriture des exercices, produire automatiquement des rapports de correction très précis et proposer de nouveaux schémas de correction pour les constructions avancées du langage. La mise en œuvre de ces outils, dans le cadre d'un cours de programmation à Sorbonne Université, est l'occasion de partager un premier retour d'expérience.

1 Introduction

Depuis la rentrée 2019, la plateforme Learn-OCaml est utilisée à Sorbonne Université dans le cadre d'un cours de remise à niveau en OCaml dispensé en première année de Master Informatique, spécialité Science et Technologie du Logiciel. Ce cours obligatoire s'adresse à un public de 58 étudiants, de niveaux très variés, sur un temps suffisamment court pour être en mesure de réaliser un projet par la suite. La préparation des séances de travaux pratiques pour ce cours a permis de tester les fonctionnalités de Learn-OCaml, et en particulier son environnement d'exercices à correction automatisée issu des technologies du MOOC OCaml [4]. Nous avons développé des outils supplémentaires [1] visant à faciliter l'écriture des exercices et rendre la correction automatisée plus précise et interactive.

Le code source d'un exercice en Learn-OCaml comprend un énoncé (`descr.md`), un environnement initial [2] (`prelude.ml` et `prepare.ml`) une esquisse de réponse destinée à l'étudiant (`template.ml`), la solution de l'enseignant (`solution.ml`) et un programme de correction (`test.ml`). La correction automatisée d'un exercice s'appuie sur le toplevel [2] dans lequel sont chargés successivement plusieurs sous-programmes, comme illustré ci-dessous.

```
⟨prelude.ml⟩ ;;
⟨prepare.ml⟩ ;;
module Code = ⟨le code de l'étudiant⟩ ;;
module Solution = ⟨solution.ml⟩ ;;
let code_ast = ⟨l'AST du module Code⟩ ;;
module Introspection = ... ;;
module Report = ... ;;
module Test_lib = ... ;;
⟨test.ml⟩ ;;
```

1. travail réalisé avec le soutien de l'IRILL (Initiative de Recherche et Innovation sur le Logiciel Libre).
2. `prelude.ml` est visible depuis le navigateur de l'étudiant, contrairement à `prepare.ml`.

Les modules `Introspection`, `Report` et `Test_lib` — fournis par Learn-OCaml — sont utilisables dans le fichier `test.ml` pour implanter le programme de correction. Le module `Introspection` offre un support pour manipuler de façon sûre le code de l'étudiant, potentiellement erroné, grâce à une extension de syntaxe permettant la représentation des types comme valeurs à l'exécution. Le module `Report` définit un type `Report.t` représentant les rapports de correction sous une forme arborescente [2]. Enfin, le module `Test_lib` propose des combinateurs de test génériques pour composer les rapports de correction.

La section 2 décrit le mécanisme d'introspection de Learn-OCaml. La section 3 présente `easy-check`, une bibliothèque que nous avons développée dans le but de faciliter l'écriture des exercices et produire automatiquement des rapports de correction très précis suivant une approche différente de celle qui est employée dans le module `Test_lib`. Cette approche nous a conduit à tester non seulement des déclarations, mais aussi des expressions quelconques, manipulant le code de l'étudiant, et pouvant donner lieu à des schémas de correction pour des traits de programmation avancés : généricité, abstraction, sous-typage, structures de données mutables et effets de bord. Enfin, la section 4 dresse un premier retour d'expérience suite à la mise en œuvre de cette bibliothèque à Sorbonne Université.

2 Introspection en Learn-OCaml

Lors de la phase de correction, le code de l'étudiant est disponible dans un module `Code`. Toutefois, si celui-ci est incompatible avec la signature attendue, l'accès à son contenu via la notation qualifiée `Code.f` provoquerait des erreurs de compilation. C'est pourquoi Learn-OCaml introduit un mécanisme d'introspection permettant d'explorer le code d'étudiant et l'exécuter partiellement sans compromettre la sûreté du typage du code de test [2].

2.1 Garder la trace des types à l'exécution

Le module `Ty` de Learn-OCaml définit un type `'a ty` représentant un type OCaml [3] sous forme d'arbre syntaxique, AST par la suite. De plus, une extension de syntaxe `[%ty: τ]` permet d'engendrer une valeur de type τ `ty` transportant précisément l'AST du type τ. Cette valeur `[%ty: τ]` appelée témoin du type τ offre alors une représentation de celui-ci à l'exécution [2]. Les phrases ci-dessous sont des exemples d'utilisation de cette extension de syntaxe.

```
# [%ty: int] ;;
- : int Ty.ty = Ty.Ty <abstr>
# [%ty: ?acc:int -> k:int -> int] ;;
- : (?acc:int -> k:int -> int) Ty.ty = Ty.Ty <abstr>

# [%ty: ([> `Cons of ('a * 'b) | `Nil] as 'a)] ;;
- : ([> `Cons of 'a * 'b | `Nil ] as 'a) Ty.ty = Ty.Ty <abstr>
# [%ty: (< me : 'a ; ..> as 'a)] ;;
- : (< me : 'a; .. > as 'a) Ty.ty = Ty.Ty <abstr>

# module type S = sig val id : 'a -> 'a end ;;
module type S = sig val id : 'a -> 'a end
# [%ty: (module S)] ;;
- : (module S) Ty.ty = Ty.Ty <abstr>

# [%ty: 'a] ;;
'a Ty.ty = Ty.Ty <abstr>
# [%ty: 'a -> 'b] ;;
- : ('_a -> 'b) Ty.ty = Ty.Ty <abstr>
# [%ty: '_a];;
Error: The type variable name '_a is not allowed in programs
```

3. Concrètement, `type 'a ty = Ty of repr and repr = Parsetree.core_type`.

2.2 Accès aux déclarations

Le module `Introspection` de Learn-OCaml fournit un support permettant d'accéder de façon sûre aux déclarations de l'environnement du toplevel.

```
type 'a value = Absent
              | Present of 'a
              | Incompatible of string

val get_value : string -> 'a Ty.ty -> 'a value
```

Accès à une variable La primitive `get_value` reçoit un identificateur `x` et un témoin de type `[%ty:` τ`]`. Si `x` est présent dans l'environnement du toplevel et lié à une valeur `(`v`:`τ'`)`, un test d'instantiation est réalisé entre τ et τ'. Si τ' est plus général que τ, la valeur retournée est `(Present` v'`)` avec v' physiquement égale à v ; dans le cas contraire, la valeur retournée est `(Incompatible` s`)` avec s un message d'erreur engendré par le compilateur. Enfin, si `x` est absent de l'environnement du toplevel, la valeur retournée est `Absent`. Les phrases ci-dessous illustrent ce principe.

```
# let x = 0 ;;
val x : int = 0
# get_value "x" [%ty: int] ;;
- : int value = Present 0
# get_value "x" [%ty: float] ;;
- : float value = Incompatible "Wrong type int."
# get_value "z" [%ty: int] ;;
- : int value = Absent

# let v = `N 0 ;;
val v : [> `N of int ] = `N 0
# get_value "v" [%ty: [> `B of bool | `N of int]] ;;
- : [> `B of bool | `N of int ] value = Present (`N 0)

# let id x = x;;
val id : 'a -> 'a = <fun>
# let f x = (match get_value "id" [%ty: 'a -> 'a] with
             | Present v -> assert (v == id); v
             | _ -> raise Not_found) x ;;
val f : 'a -> 'a = <fun>
```

Accès à un module De façon analogue, la primitive `get_value` permet d'extraire de l'environnement du toplevel un module à partir d'un identificateur de module et d'un témoin de type `[%ty: (module` S`)]`. Le module est retourné comme valeur et peut être désempaqueté.

```
# module type LIST = module type of List ;;
module type LIST = ...
# let map f l = (match get_value "List" [%ty: (module LIST)] with
                 | Present m -> let module L = (val m : LIST) in L.map
                 | _ -> raise Not_found) f l ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Un support pour l'écriture des tests L'introspection est directement utilisable dans les correcteurs automatiques pour extraire le code de l'étudiant depuis l'environnement de toplevel. Comme vu précédemment, ce mécanisme s'adapte aux constructions avancées du langage — polymorphisme paramétrique, modules, objets, sous-typage — et peut naturellement servir de support à des outils de plus haut niveau, à l'instar du module `Test_lib` de Learn-OCaml.

3 Outils et méthodologie pour l'écriture des exercices

On présente dans cette section une extension de Learn-OCaml permettant la séparation du code de correction. On s'appuie ensuite sur ce mécanisme pour developper une bibliothèque de test — qui manipule implicitement les modules `Introspection`, `Report` et `Test_lib` de Learn-OCaml — dans le but de simplifier l'écriture des exercices et annoter précisément les réponses d'étudiants. À noter que cette bibliothèque s'inspire des correcteurs automatiques de François Pottier [4] provenant du corpus d'exercices libres pour Learn-OCaml [1].

3.1 Extension du code source

Actuellement, le code de correction d'un exercice pour Learn-OCaml doit être défini dans un unique fichier `test.ml`. On ne peut pas partager de code entre les exercices, et cela les rend difficiles à écrire et à modifier. C'est pourquoi nous proposons d'introduire une courte extension dans le code source de Learn-OCaml — de l'ordre de 80 lignes de code — permettant la séparation et la réutilisation du code de correction. Cette extension offre la possibilité d'ajouter, dans le dossier d'exercices, un fichier optionnel `depend.txt`. Ce fichier indique à Learn-OCaml des noms de dépendances `.ml` et `.mli` à charger dynamiquement dans le toplevel pendant la phase de correction, avant que le fichier `test.ml` ne soit évalué. Ainsi, le code de correction dans `test.ml` a non seulement accès aux modules de Learn-OCaml, mais aussi à des modules de dépendances déclarés par l'enseignant.

3.2 Une nouvelle bibliothèque de test

L'extension présentée ci-dessus a motivé le développement d'`easy-check`, une bibliothèque de test utilisable comme dépendance pour implanter les exercices. Cette bibliothèque comprend environ 1000 lignes de code réparties en 10 modules parmi lesquels `Get`, `Result`, `Assume`, `Check` et `Autotest`, succinctement décris par la suite.

Des annotations concises La stratégie mise en œuvre dans `easy-check` pour annoter les programmes d'étudiants repose sur trois modèles de rapports de correction :

- $\mathbf{I}_x$ indique que la déclaration x est absente ou incompatible avec la signature attendue ;
- $\mathbf{F}_e$ donne un contre-exemple justifiant que l'expression `Code.(`e`)` est incorrecte ;
- $\mathbf{T}_e$ indique que l'expression `Code.(`e`)` semble correcte.

Structure d'un fichier `test.ml` Un fichier `test.ml` basé sur `easy-check` contient en général un code de correction de la forme `(Result.set [`q_1`;`q_2`;...])` où chaque q_i est une fonction de type `unit -> Report.t` qui peut signaler une exception `Fail of Report.t`. Le combinateur `Result.set` calcule chaque expression `(try` q_i `() with Fail` r `->` r`)`, fusionne les rapports de correction obtenus, puis renvoie le résultat à l'étudiant.

Accès au code de l'étudiant L'expression `(Get.value "x" [%ty:` τ`])` extrait la variable `(Code.x :` τ`)` de l'environnement du toplevel, grâce à `Introspection.get_value`. Si `Code.x` est absente ou incompatible avec le type τ, alors l'exception `(Fail` $\mathbf{I}_x$`)` est signalée. De manière analogue, `(Get.mod_value "M" [%ty: (module` S`)])` extrait le module `M` de l'environnement du toplevel et le restitue sous une forme empaquetée. Enfin, `(Get.code [%ty: (module` S`)])` est une abréviation pour `(Get.mod_value "Code" [%ty: (module` S`)])`.

4. https://github.com/ocaml-sf/learn-ocaml-corpus/blob/master/exercises/merge_sort/test.ml

Vérifier le type d'une déclaration L'expression `(Assume.compatible "x" [%ty: τ])` rend `()` si `Code.x` est bien présente dans le toplevel et si son type est plus général que τ. Sinon, l'exception `(Fail I`$_x$`)` est signalée.

Tester une déclaration L'expression `(Check.name`$_n$ `"f" [%ty:` τ_1 `->` ... `->` τ_n `->` τ_{res}`] ℓ)` accède via l'introspection à $f_{\texttt{code}}$ = `(Code.f :` τ_1 `->` ... `->` τ_n `->` τ_{res}`)`, puis évalue l'expression `(`$f_{\texttt{code}}$ a_1 ... a_n`)` pour chaque n-uplet `(`a_1`,...,`a_n`)` pris dans la liste ℓ. Chaque valeur retournée est comparée vis-à-vis de `(Solution.f` a_1 ... a_n`)`. Si un contre-exemple est trouvé, alors l'exception `(Fail F`$_f$`)` est signalée. Sinon, le rapport `T`$_f$ est retourné. La comparaison entre `Code.f` et `Solution.f` est personnalisable via différents arguments optionnels de `Check.name`$_n$. La valeur de l'argument `~equal` (respectivement `~equal_exn`) compare [5] les valeurs retournées (respectivement les exceptions signalées) par `Code.f` et `Solution.f`. Par ailleurs, la valeur de l'argument `~equal_stdout` (respectivement `~equal_stderr`) compare [6] les affichages produits par `Code.f` et `Solution.f` sur la sortie standard (respectivement la sortie d'erreur). Ainsi, le fichier `test.ml` suivant teste une fonction d'affichage (`q1`), une fonction d'arité 2 (`q2`) et une fonction polymorphe (`q3`). On constate que, pour tester une fonction polymorphe, il suffit de vérifier son type, puis tester une instance de la fonction.

```
let q1 () = Check.name1 "print_int_list" [%ty: int list -> unit]
                        ~equal_stdout:(=) [[];[0];[1;2];...] ;;
let q2 () = Check.name2 "pow" [%ty: int -> int -> int] [(2,8);(4,3);...] ;;
let q3 () = Assume.compatible "sort" [%ty: 'a list -> 'a list];
            Check.name1 "sort" [%ty: int list -> int list] [[2;3;1];...] ;;
let () = Result.set [q1;q2;q3] ;;
```

Tester une expression En Learn-OCaml, on est souvent amené à vouloir tester, non seulement des déclarations, mais aussi des expressions manipulant le code de l'étudiant. Supposons par exemple que l'on souhaite tester un opérateur `(|>) : 'a -> ('a -> 'b) -> 'b` défini par l'étudiant. Une solution naturelle consiste à tester une expression de la forme `Code.(`e `|>` f`)`. Toutefois, à notre connaissance, il est difficile d'écrire un tel code de correction avec le module `Test_lib` de Learn-OCaml. C'est pourquoi `easy-check` propose des combinateurs supplémentaires, qui généralisent `Check.name`$_n$ dans le but de tester des expressions quelconques.

Pour cela, nous introduisons d'abord l'extension de syntaxe [7] `[%code` e`]` qui expanse le triplet `(Code.(`e`), Solution.(`e`),` e_p`)` avec e_p une chaîne de caractères qui représente l'expression e en syntaxe OCaml [8] en vue de l'afficher dans le rapport de correction.

L'expression `(Check.expr`$_n$ `[%code` e`] [%ty:` τ_1 `->` ... `->` τ_n `->` τ_{res}`] ℓ)` évalue l'expression `((Code.(`e`)` a_1 ... a_n`) :` τ_{res}`)` pour chaque n-uplet `((`a_1`,...,`a_n`) : (`τ_1 `*` ... `*` τ_n`))` pris dans la liste ℓ. Chaque valeur retournée est comparée vis-à-vis de `(Solution.(`e`)` a_1 ... a_n`)`. Si un contre-exemple est trouvé, alors l'exception `(Fail F`$_e$`)` est signalée. Sinon, le rapport `T`$_e$ est retourné. Le fichier `test.ml` suivant teste l'opérateur `Code.(|>)` comme évoqué plus haut.

```
module type CODE = sig val (|>) : 'a -> ('a -> 'b) -> 'b end ;;
let q1 () =
  let m = Get.code [%ty: (module CODE)] in
  let module Code = (val m : CODE) in
  Check.expr1 [%code (fun n -> n + 1 |> string_of_int)]
              [%ty: int -> string] [1;2;3;4] ;;
let () = Result.set [q1] ;;
```

5. Par défaut, c'est l'égalité structurelle d'OCaml qui est utilisée.
6. Si non précisé, les affichages sont ignorés.
7. Cette extension ppx nécessite un ajout, d'environ 20 lignes, dans le code source de Learn-OCaml.
8. Cette chaîne de caractères est produite directement à partir de l'AST de l'expression e, au moyen de la fonction `Pprintast.string_of_expression` de la distribution OCaml.

À noter que l'extension de syntaxe `[%code e]` manipule le module `Code` de façon non sûre. En particulier ci-dessus, la valeur `[%code (fun n -> n + 1 |> string_of_int)]` n'est correcte que si `Code.(|>)` est compatible avec le type `int -> (int -> string) -> string`. En effet, si la réponse de l'étudiant n'est pas compatible avec la signature attendue, une erreur de compilation surviendrait lors du chargement de fichier `test.ml` dans le toplevel. Il est donc indispensable de redéfinir localement un module `Code` sûr grâce à `Get.code`. Ce recours à l'introspection peut sembler laborieux au premier abord. Pour autant, les rapports de correction engendrés par `Check.expr`$_n$ sont alors très précis, concis et plus expressifs :

```
The following expression:                                    1 pt
  (fun n  -> (n + 1) |> string_of_int)
seems correct.
```

Engendrer les valeurs pour les tests L'argument optionnel `~testers` de `Check.name`$_n$ et `Check.expr`$_n$ attend une liste de valeurs de type `(`τ_1`,`$\tau_2`) tester`. Le rôle d'un testeur est d'engendrer des valeurs x_i de type τ_1 puis leur appliquer la fonction candidate f `:` τ_1 `->` τ_2 et une fonction solution f'. Les deux résultats sont alors comparés à l'aide d'une fonction eq `:` τ_2 `->` τ_2 `-> bool` afin de mettre en évidence un éventuel contre-exemple $\langle x_0, v, v' \rangle$ tel que `(`f x_0`)` s'évalue en v, `(`f' x_0`)` en v' et `(`eq v v'`)` en $false$. Le module `Autotest` d'`easy-check` fournit un testeur paramétré `Autotest.tester` qui, étant donnée une valeur de type τ_1 `sampler = unit ->` τ_1, rend un testeur de type `(`τ_1`,'a) tester`. Ce module propose également des utilitaires [9] pour composer des sampleurs. `(fun` x `->` `Autotest.choose` f g x`)`, par exemple, engendre soit `(`f x`)`, soit `(`g x`)`. `(Autotest.oneof [`x_1`;...;`x_n`])` engendre une valeur prise parmi x_1, ..., x_n. `(Autotest.tree ~depth:`d f `(fun t ->` g `t))` engendre un arbre de profondeur `(`d `())`, de feuilles `(`f `())` et de nœuds `(`g `t)` avec `(t ())` un sous-arbre.

Un exemple d'exercice : réduction de λ-termes On propose en exercice la définition d'une fonction `reduce` qui réduit un terme du λ-calcul pur suivant une stratégie d'appel par valeur. On représente les termes par le type `t = Var of var | Lam of (var * t) | App of (t * t) and var = string`. Voici le code de correction (fichier `test.ml`) :

```
1  let lambda_term v =
2    let lam t = Lam (v (),t ())
3    and app t = App (t (),t ()) in
4    Autotest.(tree ~depth:(nat 5) (fun () -> Var (v ())) (choose lam app));;
5  let q1 () =
6    Check.name1 "reduce" [%ty: t -> t]
7                ~testers: [Autotest.(tester (lambda_term (oneof ["x";"y";"z"])))]
8                [] ;;
9  let () = Result.set [q1];;
```

On teste la fonction `reduce` avec `Check.name1` qui accède de façon sûre à la variable `Code.reduce` et l'applique à des termes engendrés automatiquement par `Autotest.tree` pour mettre en évidence un contre-exemple. Si l'étudiant propose une solution incorrecte, un rapport annoté lui indiquera clairement l'origine de son erreur ; ici une capture de variable :

```
The following expression:                                    0 pt
  reduce (App (Lam ("z", App (Lam ("z", Var "z"), Var "x")), Var "y"))

... produces the following value:
  (Var "y")
This is incorrect. Producing the following value is correct:
  (Var "x")
```

9. Les sampleurs d'`Autotest` suivent le modèle du module `Test_lib` de Learn-OCaml, qui fournit « des combinateurs de génération aléatoire pour les principaux types prédéfinis [...] avec différents paramètres permettant de biaiser la distribution » [2]. À l'avenir, on pourrait reprendre les travaux de génération aléatoire de structures de données [3] et ainsi s'assurer que la génération est uniforme.

Modules et types abstraits Une problématique récurrente en Learn-OCaml est la correction des fonctions manipulant des types définis par l'étudiant : « Soit un module avec un type abstrait `t` et deux fonctions `to_string : t -> string` et `from_string : string -> t`. Tester ces deux fonctions n'est pas possible puisqu'elles utilisent des valeurs d'un type `t` qui peut être implémentés différemment par l'étudiant et par l'enseignant. Une solution consiste à simplement tester `(fun s -> to_string (from_string s))` » [4]. Le fichier `test.ml` ci-dessous illustre la mise en œuvre de cette technique avec `easy-check`. On redéfinit un module `Code` sûr grâce à `Get.code`, puis à l'aide de notre extension de syntaxe `[%code` e`]`, on teste l'expression souhaitée en l'appliquant à des chaînes de caractères engendrées par `Autotest.string`.

```
1  module type CODE = sig
2    module M : sig
3      type t
4      val from_string : string -> t
5      val to_string : t -> string
6    end
7  end ;;
8  let q1 () = let m = Get.code [%ty: (module CODE)] in
9              let module Code = (val m : CODE) in
10             Check.expr1 [%code (fun s -> M.to_string (M.from_string s))]
11                         [%ty: string -> string]
12                         ~testers: [ Autotest.(tester string) ]
13                         [""] ;;
14 let () = Result.set [q1] ;;
```

3.3 Méthodologie pour l'écriture des exercices

Le support que nous avons proposé pour la séparation du code de correction, au moyen d'une courte extension dans le code source de Learn-OCaml, rend possible une certaine discipline de programmation. Un point clé est le fait de pouvoir s'appuyer sur des fichiers de dépendances qui, puisqu'ils sont chargés dynamiquement dans le toplevel, peuvent être modifiés sans devoir recompiler Learn-OCaml. Des barrières d'abstraction peuvent alors être mises en place, masquant les primitives d'introspection et la représentation concrète des rapports de correction pour se focaliser sur les seuls aspects pédagogiques. L'expérimentation est alors facilitée et encouragée, à l'instar de notre bibliothèque de test, qui annote avec précision la réponse de l'étudiant, de manière automatique, par la mise en évidence d'un éventuel contre-exemple indiquant clairement ce qui est incorrect et ce qui est attendu. La continuation de cette approche consiste à tester des fragments de programmes, autrement dit des expressions, au moyen d'une extension de syntaxe `[%code` e`]`. On en déduit de nouveaux schémas de correction pour les constructions avancées du langage, telles que les modules paramétrés ou les objets.

trait du langage	signature du module `Code`	expression à tester
module	`sig module M : S end`	`[%code (let open M in` e`)]`

foncteur	`sig module F :`
	`  functor (X:S1) -> S2`
	`end`
		`[%code`
		`  (let module M = F(V) in`
		`  let open M in` e`)]`

opérateur	`sig val (!) =` τ_1 `->` τ_2 `end`	`[%code (!)]`
appel de méthode	`sig val o : < m :` τ `> end`	`[%code (o#m)]`

classe	`sig class ['a] c =`
	`  object method m :` τ `end`
	`end`
		`[%code ((new c)#m)]`

classe abstraite	`sig class virtual c = object`
	`  method virtual m1 :` τ_1 `end`
	`  method m2 :` τ_2 `end`
	`end`
		`[%code`
		`(let o = object inherit c`
		`  method m1 =` v `end in`
		`  e)]`

4 Conclusion

Cet article présente `easy-check`, la bibliothèque que nous avons développée pour faciliter l'écriture d'exercices à correction automatisée en Learn-OCaml. Cet outil a été mis en œuvre dans l'UE « Ouverture » dispensée à l'ensemble des étudiants de M1 Informatique spécialité Science et Technologie du Logiciel à Sorbonne Université. Au regard de la diversité des profils d'étudiants, il apparaît que Learn-OCaml permet à chacun de progresser à son propre rythme, en choisissant les exercices adaptés parmi des séries d'exercices de difficultés variées. On a de plus constaté que Learn-OCaml pousse les étudiants à travailler de manière plus autonome, tout en interagissant individuellement avec l'enseignant, ce qui est propice à un apprentissage efficace en séances de travaux pratiques comme cela avait été formulé aux JFLA l'an passé [1]. Enfin, l'accès direct à l'environnement via un navigateur Web est un gain de temps précieux, car cela évite l'installation d'un environnement OCaml classique pendant la séance de travaux pratiques [5]. C'est pourquoi on envisage d'adopter Learn-OCaml dès le semestre prochain. On pense en particulier au nouveau cours de L2 « Programmation fonctionnelle », et à celui de L3 dont sont tirés la plupart des exercices de cette expérimentation.

Des correcteurs automatiques ont déjà été utilisés à Sorbonne Université ; en particulier l'environnement polyglotte CodeGradX [6] développé par Christian Queinnec, pour des examens de Licence en OCaml et Java. On remarquait que, plus les annotations automatiques sur les copies étaient précises, plus la correction était acceptée, car les étudiants lisaient vraiment ces annotations et posaient alors des questions pertinentes. L'environnement MrPython — utilisé pour le cours d'initiation à la programmation en première année de Licence à Sorbonne Université — propose par ailleurs un cadre pour évaluer les tests effectués par les étudiants. Une démarche similaire a été employée dans un cours de programmation en OCaml à l'Université McGill : demander aux étudiants d'écrire leurs propres tests et utiliser l'infrastructure de correction automatique de Learn-OCaml pour évaluer leur pertinence [5].

Plusieurs projets collaboratifs s'appuient sur Learn-OCaml [1] pour promouvoir l'enseignement du langage OCaml. Il s'agit maintenant de pouvoir les combiner.

Références

[1] Çağdaş Bozman, Benjamin Canou, Roberto Di Cosmo, Pierrick Couderc, Louis Gesbert, Grégoire Henry, Fabrice le Fessant, Michel Mauny, Carine Morel et Loïc Peyrot : Learn-OCaml : un assistant à l'enseignement d'OCaml. *In Trentièmes Journées Francophones des Langages Applicatifs (JFLA'19)*, janvier 2019.

[2] Benjamin Canou, Çağdaş Bozman et Grégoire Henry : Sous le capot du MOOC OCaml. *In Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA'16)*, janvier 2016.

[3] Benjamin Canou et Alexis Darrasse : Fast and sound random generation for automated testing and benchmarking in Objective Caml. *In ACM SIGPLAN Workshop on ML*, pages 61–70, 2009.

[4] Benjamin Canou, Roberto Di Cosmo et Grégoire Henry : Scaling up functional programming education : under the hood of the OCaml MOOC. *In ACM SIGPLAN International Conference on Functional Programming (ICFP'17)*, pages 1–25, août 2017.

[5] Aliya Hameer et Brigitte Pientka : Teaching the art of functional programming using automated grading (experience report). *In ACM SIGPLAN International Conference on Functional Programming (ICFP'19)*, août 2019.

[6] Christian Queinnec : An Infrastructure for Mechanised Grading. *In 2nd International Conference on Computer Supported Education (CSEDU'10)*, pages 37–45, avril 2010.

Détection de définitions OCaml similaires (ou comment ne plus voir double à dos de chameau)

Alexandre Moine et Yann Régis-Gianas

Université de Paris (UMR-CNRS 7126) - INRIA PI.R2

Résumé

L'absence de redondance est souvent un gage de qualité pour un code source. En effet, lorsqu'un fragment de code est répété, ses imperfections – pour ne pas dire ses erreurs – le sont elles-aussi.

Seulement, il arrive parfois que l'on constate de la redondance dans un grand corpus de code, typiquement quand ce corpus a été construit par des développeurs ne communiquant peu ou pas du tout entre eux. Deux instances de cette situation nous intéressent particulièrement : l'ensemble des codes sources des paquets OPAM et l'ensemble des copies d'étudiants répondant tous aux mêmes questions de programmation. Comment partitionner leurs définitions en fonction de leur "similarité" ?

Dans cet article, nous proposons un outil de partitionnement automatique d'un ensemble de définitions écrites en OCaml. Cet outil s'appuie sur une fonction dédiée de prise d'empreintes des arbres de syntaxe du langage intermédiaire Lambda ainsi que sur un algorithme de classification hiérarchique classique que nous avons adapté à notre usage.

Cet outil prend la forme d'une bibliothèque nommée Asak disponible sur OPAM. Nous l'avons utilisée d'une part pour partitionner automatiquement les réponses d'étudiants qui apprennent OCaml en utilisant la plateforme LearnOCaml, et d'autre part, pour détecter des redondances sur l'ensemble des codes sources des paquets OPAM disponibles aujourd'hui. Nous évaluons les résultats obtenus et formulons quelques limites de notre approche.

1 Introduction

"Ne vous répétez pas !" est une injonction permanente qui plane au dessus de tout programmeur cherchant à écrire un logiciel de qualité [10]. Ce principe vise une situation idéale où une connaissance donnée n'a qu'une seule représentation dans le logiciel : si cette connaissance est imparfaite – et c'est souvent le cas – on s'assure alors qu'elle pourra être corrigée efficacement.

Il arrive parfois qu'un corpus de code source contienne des redondances que les programmeurs n'ont pas pu ou pas voulu éviter. Par exemple, certaines fonctions utilitaires absentes de la bibliothèque standard sont ainsi réimplémentées à de multiples reprises par de nombreux paquets logiciels. L'implémentation de ces fonctions utilitaires répétées sont parfois identiques aux caractères près ; parfois, elles sont seulement très proches (à quelques renommages près ou plus généralement, à quelques réécritures "bénignes" près). Ainsi, si l'on utilise deux bibliothèques indépendantes pour construire un logiciel donné, il y a une probabilité non nulle que l'on retrouve dans l'exécutable final plusieurs implémentations quasi-identiques de la même fonction. Ne serait-il pas plus raisonnable d'introduire une bibliothèque commune offrant une unique implémentation de ces fonctions répétées ?

Il y a aussi des redondances dont les programmeurs n'ont pas conscience. En effet, on peut parfaitement appliquer "la tête dans le guidon" des schémas de calcul généraux en les spécialisant à des arguments particuliers, au cas par cas, sans réaliser les opportunités de factorisation qui découleraient de l'introduction d'une fonction d'ordre supérieur bien choisie. C'est seulement

lorsque les développeurs prennent un peu de recul qu'ils s'aperçoivent qu'une factorisation du code source est possible. Ne serait-il pas plus efficace d'alerter le programmeur à l'instant même où il introduit un fragment de code qui est fortement similaire à un autre fragment préexistant, soit dans son propre code, soit dans une autre bibliothèque ?

Lorsqu'un corpus est formé de réponses d'étudiants à une même question de programmation, il y a fort à parier que plusieurs étudiants répondront de façon similaire à une question donnée, soit parce qu'ils ont trouvé une réponse "naturelle", soit parce qu'ils ont commis des erreurs de raisonnement ou de conception classiques. Pour l'enseignant face à plusieurs centaines de réponses, il n'est pas toujours évident de réaliser quelles sont les classes qui partitionnent pertinemment l'ensemble de ces réponses. Lui apporter un tel partitionnement serait un outil précieux car il lui permettrait de différencier sa pédagogie en fonction des groupes de réponses fausses les plus courantes.

Pour répondre à ces trois cas d'usage, il faudrait tout d'abord que l'on sache évaluer la "similarité" entre deux fragments de code. Mais qu'entend-on exactement par similarité ? Peut-on formaliser cette notion et la réaliser calculatoirement ?

Notons tout d'abord que nous avons utilisé le terme de similarité et non d'équivalence. En effet, la notion d'équivalence (syntaxique, définitionnelle ou observationnelle) est trop "binaire" pour notre cadre : nous cherchons une mesure qui rapproche des calculs qui se ressemblent même s'ils ne se comportent pas tout à fait de la même façon calculatoirement. En d'autres termes, nous cherchons des fragments de programme qui ont des structures syntaxiques proches et qui s'appuient sur des ingrédients similaires plutôt que des fragments de programme ne pouvant pas être distingués par des contextes d'évaluation.

Ainsi, deux termes égaux syntaxiquement sont absolument similaires. Deux termes qui diffèrent par un renommage sont très similaires sans être syntaxiquement égaux. Deux analyses de motifs sont plus ou moins similaires en fonction des cas d'analyse qu'elles traitent de façon similaire. Par contre, les implémentations de deux algorithmes de tri distincts sont en général dissimilaires même si l'équivalence observationnelle ne permet pas de les distinguer [1].

La similarité semble donc être une notion très syntaxique mais qui cherche paradoxalement à faire peu de cas de certains détails d'écriture qui ne changent pas fondamentalement le programme. De toute évidence, toutes les différences purement textuelles (commentaires, indentations, ...) doivent être ignorées par une bonne notion de similarité. Le renommage des variables liées est aussi un exemple canonique de tels détails syntaxiques anodins. Plus généralement, toute transformation locale et purement syntaxique, i.e. toute élimination de sucre syntaxique, semble aussi rentrer dans cette catégorie des "détails syntaxiques". Où s'arrêter dans ce processus de simplification des termes sources ? Devrait-on par exemple aller jusqu'à comparer les codes machines obtenus par compilation des termes sources ? Cette démarche conduirait sans doute à un échec puisque le jeu de la sélection d'instructions et des différentes optimisations peut mener deux termes sources proches à des codes machines très différents et fourmillant de nouveaux détails peu importants (pensez à la diversité des instructions d'une architecture comme x86-64). Il faut donc trouver le langage intermédiaire offrant un bon niveau d'abstraction pour éliminer à la fois les détails syntaxiques du langage source et les détails de bas-niveau de l'architecture cible.

La première contribution de cet article est de considérer que les premières passes du compilateur OCaml, celles menant de sa syntaxe concrète au code Lambda, éliminent la plupart des détails syntaxiques des termes OCaml pour ne garder que leurs ingrédients calculatoires

1. Bien entendu, on suppose ici un langage de programmation incapable d'observer finement les exécutions des deux algorithmes.

principaux. Nous analyserons les avantages (sections 5 et 6) et les limitations (section 7) de ce choix de conception.

Comme son nom l'indique, LAMBDA est un λ-calcul (étendu). Nous nous sommes donc moralement ramené au problème de la construction d'une mesure de similarité syntaxique entre deux termes du λ-calcul. En utilisant une représentation de De Bruijn et en effectuant quelques réductions inoffensives, on gomme effectivement de cette façon certaines différences sans intérêt entre deux termes OCAML. Seulement, pour pouvoir comparer rapidement des milliers de $\lambda-$termes deux-à-deux, on doit se donner une représentation compacte des λ-termes. Nous introduisons donc un prétraitement des λ-termes pour calculer leurs *empreintes* respectives : l'empreinte d'un λ-terme est un ensemble d'entiers qui caractérisent ses aspects syntaxiques importants. C'est une idée déjà présente dans la littérature [5] mais nous la raffinons pour l'appliquer à des λ-termes. La définition de ce prétraitement sur des λ-termes est donc la seconde contribution présentée par cet article.

À ce stade, on peut donc mesurer la similarité entre chaque paire de termes d'un corpus. L'ensemble de ces mesures constitue une grande quantité d'information peu structurée qui est donc difficilement exploitable directement, en tout cas, pour nos cas d'usage. Pour pallier à ce problème, nous proposons de partitionner hiérarchiquement les termes en s'appuyant sur leur mesure de dissimilarité (section 4). Encore une fois, notre contribution consiste à raffiner et implémenter un algorithme déjà présent dans la littérature, la classification ascendante hiérarchique. Cette classification produit des dendrogrammes, particulièrement adaptés à l'exploration progressive des classes de programmes issues de nos corpus. Un dendogramme est un arbre binaire dont les nœuds représentent des classes d'individus.

Pour que ce travail soit réutilisable dans des situations différentes de nos propres cas d'usage, nous avons développé une bibliothèque nommée ASAK [2]. Cette bibliothèque est librement disponible sous la forme d'un paquet OPAM et sur GITHUB [1]. Elle nous a permis d'introduire un système de classification automatique des copies dans LEARNOCAML [3] (section 5). Elle nous a aussi permis d'implémenter un outil de recherche de redondance dans l'ensemble des paquets OPAM (section 6). Ces outils constituent la dernière contribution présentée par cet article.

Cet article décrit la première version d'ASAK : il s'agit d'un travail de recherche en cours dont les résultats préliminaires nous ont semblé suffisamment intéressants pour être communiqués. Il reste encore beaucoup à faire pour rendre l'outil plus performant et plus pertinent. Nous évaluons ses limitations (section 7), le comparons à l'état de l'art (section 8) et donnons les pistes à explorer pour tenter de les surmonter (section 9).

2 Vue d'ensemble de l'approche

Que produit ASAK ? ASAK compare entre elles les définitions globales d'un ensemble de modules OCAML. La figure 2 contient cinq exemples de telles définitions OCAML. Elles forment un corpus jouet qui va nous servir à illustrer le fonctionnement d'ASAK. Le lecteur aura reconnu d'un coup d'œil la fonction classique qui renvoie la liste miroir d'une liste prise en argument et l'enseignant aura reconnu des réponses typiques d'étudiants apprenti-programmeurs fonctionnels. Sur ce corpus, notre outil produit l'ensemble de dendrogrammes de la figure 2.

Un dendrogramme est un arbre binaire dont les nœuds représentent des classes d'individus. Les feuilles de ces dendrogrammes sont les différentes versions de `rev`. Un dendrogramme fournit un partitionnement hiérarchique d'un ensemble d'individus : en le parcourant d'une feuille vers

2. Cette bibliothèque fait des partitions (de codes similaires), son nom est donc tiré de la musique : ASAK est un genre de chansons touareg.

```
(* Code 1 *)
let rec rev l = match l with
    [] -> []
  |t::q -> rev q@[t]
(* Code 2 *)
let rec rev l =
  match l with
  | [] -> []
  | a::t -> (rev t)@[a]
(* Code 3 *)
let rec rev l = match l with
    [] | [_] -> l
  | t::q -> rev q@[t];;
```

```
(* Code 4 *)
let rev l=
  let rec rev_aux acc l=
    match l with
    |[]->acc
    |t::q->rev_aux (t::acc) q
  in rev_aux [] l
(* Code 5 *)
let rev l =
  match l with
    [] -> []
  |a::q -> let rec rev2 x y = match y with
         [] -> x
       |b::z -> rev2 (b::x) z in rev2 [] l
(* Code 6 *)
let rev l =
  List.fold_left (fun acc x -> x :: acc) [] l
```

FIGURE 1 – Un corpus jouet pour illustrer notre algorithme.

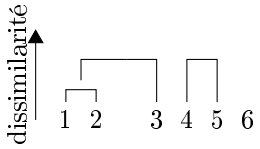


FIGURE 2 – Ensemble de dendrogrammes pour la classification du corpus jouet.

sa racine, on découvre des classes d'invididus de plus en plus dissimilaires à cette feuille ; en le parcourant de sa racine vers ses feuilles, on découvre des séparations successives en deux classes d'un ensemble d'individus maximisant la dissimilarité entre les membres des deux classes. Dans la sortie de notre outil, deux classes qui appartiennent à deux dendrogrammes distincts sont absolument dissimilaires, i.e. elles n'ont rien en commun.

Dans le cas de ce corpus, le regroupement des définitions 1 et 2 est naturel puisque ces deux définitions sont équivalentes à quelques détails purement textuels près : la présence du |, d'un retour à la ligne et d'un renommage. La définition 3 est proche de cette première classe : elle en diffère seulement à cause d'un cas d'analyse supplémentaire. Le regroupement des définitions 4 et 5 est assez naturel lui aussi puisqu'elles suivent la même (et bonne !) stratégie qui consiste à accumuler le résultat dans l'argument d'une fonction auxiliaire interne. Notons que ces deux définitions sont dissimilaires même si elles utilisent les mêmes "ingrédients" : la définition 5 effectue une analyse par cas supplémentaire pour traiter la liste vide de façon spécifique.

Comment procède ASAK ? Ce partitionnement semble pertinent mais comment notre outil l'a-t-il obtenu ? Comme nous l'avons déjà écrit dans l'introduction, le traitement de notre outil se décompose essentiellement en trois étapes : (i) les programmes sources sont normalisés pour ignorer les détails syntaxiques que nous jugeons inessentiels ; (ii) on calcule une empreinte pour caractériser la structure et les ingrédients principaux des programmes normalisés ; (iii) on applique un algorithme de partitionnement hiérarchique qui s'appuie sur les empreintes.

```
Définition 1: (function l/88 (if l/88
    (apply (field 36 (global Stdlib!)) (apply rev/87 (field 1 l/88))
      (makeblock 0 (field 0 l/88) 0a)) 0a))
Définition 2: (function l/88 (if l/88
    (apply (field 36 (global Stdlib!)) (apply rev/87 (field 1 l/88))
      (makeblock 0 (field 0 l/88) 0a)) 0a))
Définition 3: (function l/88 (catch
    (if l/88
      (if (field 1 l/88)
        (apply (field 36 (global Stdlib!)) (apply rev/87 (field 1 l/88))
          (makeblock 0 (field 0 l/88) 0a))
        (exit 12))
      (exit 12)) with (12) l/88))
Définition 4: (function l/88 (letrec (rev_aux/89
       (function acc/90 l/91
         (if l/91
           (apply rev_aux/89 (makeblock 0 (field 0 l/91) acc/90)
             (field 1 l/91))
           acc/90))) (apply rev_aux/89 0a l/88)))
Définition 5: (function l/88 (if l/88
    (letrec (rev2/91
         (function x/92 y/93
           (if y/93
             (apply rev2/91 (makeblock 0 (field 0 y/93) x/92) (field 1 y/93))
             x/92)))
      (apply rev2/91 0a l/88)) 0a))
Définition 6: (function l/88 (apply (field 20 (global Stdlib__list!))
    (function acc/146 x/147 (makeblock 0 x/147 acc/146)) 0a l/88))
```

FIGURE 3 – Traduction du corpus dans le code LAMBDA du compilateur OCAML.

Comment les définitions sont-elles normalisées ? L'analyse syntaxique est la solution canonique pour éliminer les artefacts textuels et ne garder que la structure d'un code source. Notre outil travaille donc sur des arbres de syntaxe abstraits.

Une fois que l'on a décidé de travailler sur un langage d'arbres distincts de la syntaxe concrète, il faut choisir ce nouveau langage. On aurait pu choisir de traduire les termes sources vers un langage conçu pour l'occasion mais ce serait beaucoup de travail. Il existe heureusement un langage intermédiaire adéquat dans le compilateur OCAML : le langage LAMBDA. Nos expérimentations nous portent à croire qu'il se place au bon niveau d'abstraction pour capturer l'essence de la structure calculatoire du programme source.

Ce langage sera présenté précisément dans la section 3.1 mais nous pouvons d'ores et déjà donner la traduction du corpus jouet dans la figure 3. Pour réaliser cette traduction, nous réutilisons la partie avant du compilateur OCAML et nous effectuons un post-traitement qui normalise encore un peu plus les termes. Les détails de cette traduction seront décrits dans la section 3.1. Sur nos exemples, on peut déjà remarquer que les définitions 1 et 2 sont identiques une fois normalisées. On note aussi que la définition 3 normalisée partage des sous-termes avec les définitions 1 et 2 normalisées. Des remarques similaires s'appliquent aux autres définitions.

Pourquoi calcule-t-on des empreintes ? Comparer deux arbres en itérant sur leurs structures respectives a un coût proportionnel à leur taille. Par ailleurs, la classification qui nous

intéresse doit idéalement savoir comparer l'ensemble des sous-arbres des deux arbres pour déterminer quelle quantité de code ils ont en commun. L'ordre d'apparition des sous-arbres n'est donc pas forcément important : bien entendu, deux arbres utilisant les mêmes sous-arbres dans le même ordre seront très similaires mais utiliser les mêmes sous-arbres dans un ordre différent est aussi une forme de similarité non négligeable même si elle est un peu moins forte.

Après ces remarques, l'implémentation d'une fonction d'évaluation de la similarité entre deux termes Lambda semble difficile. Nous introduisons les empreintes d'arbres pour simplifier cette implémentation et aussi pour la rendre efficace. Les empreintes sont des ensembles de clés de hachages des sous-termes (suffisamment gros) du terme Lambda. L'idée importante de cette notion d'empreinte est de prendre en compte l'ordre relatif des sous-termes dans le calcul de la clé de hachage tout en regardant aussi l'empreinte comme un ensemble de clés pour maintenir une certaine proximité entre les termes qui utilisent les mêmes sous-termes, mais dans un ordre différent. Ainsi, les deux programmes suivants :

```
1 let f () = e1; e2
2 let g () = e2; e1
```

ont pour empreintes :

$$\begin{array}{rcl} E(\texttt{f}) & = & \{H(\texttt{e1;\ e2}), H(\texttt{e1}), H(\texttt{e2})\} \\ E(\texttt{g}) & = & \{H(\texttt{e2;\ e1}), H(\texttt{e1}), H(\texttt{e2})\} \end{array}$$

où $E(t)$ est l'empreinte de t et $H(t)$ est la clé de hachage de t.

Ces deux empreintes sont distinctes mais partagent deux clés de hachage. Ce partage témoigne du fait qu'elles "utilisent les mêmes ingrédients". Les empreintes calculées pour les définitions de notre corpus jouet se trouvent dans la figure 4. On distingue la liste numérotée des clés de hachage dans la partie haute de la figure. En bas de la figure, on trouve les empreintes des définitions, ce sont des (multi-)ensembles de paires d'entiers. Dans cette figure, l'entier du haut est le numéro de clé de hachage et l'entier du bas est le nombre de nœuds de l'arbre de syntaxe du sous-terme haché. Nous donnons la définition précise de cette prise d'empreintes dans la section 3.

Comment le partitionnement hiérarchique est-il effectué ? Il existe deux grandes familles de partitionnement hiérarchique : les partitionnements ascendants et les partitionnements descendants. Les partitionnements ascendants sont adaptés aux corpus formés de petits groupes tandis que les partitionnements descendants à ceux formés de grands groupes. Nous avons fait l'hypothèse que les groupes de définitions similaires sont petits par rapport au corpus analysé et nous avons donc choisi un algorithme de partitionnement hiérarchique ascendant.

L'algorithme procède donc en partant d'un partitionnement où chaque individu est dans une classe distincte de celles des autres et choisit à chaque itération de fusionner les deux classes les moins dissimilaires. Nous définirons précisément la mesure de dissimilarité que nous utilisons dans la section 4 mais pour se donner une idée du fonctionnement de l'algorithme, le lecteur pourra en observer la trace d'exécution sur notre corpus jouet dans la figure 5.

3 Prise d'empreintes

3.1 Lambda normalisé

Présentation de Lambda La syntaxe du langage Lambda est donnée dans la figure 6. Il s'agit d'un λ-calcul avec quelques spécificités par rapport à celui que l'on trouve dans les présentations à saveur plus théoriques. Tout d'abord, les fonctions ne sont pas unaires : elles ont une arité potentiellement supérieure à 1. Ensuite, on distingue les primitives des constantes :

01 = 6cdbdd45d9fda8de7da9dc4d9add5d43d4ad80d44df9d9cd
02 = 58de8dc8d46d83d87d4bd23d24d93d82d69de2d32d64dc6d
03 = dddc1d3dd8adf7d7d41d4dd8bd9d2ed30d6edf4d9fd2d
04 = 64dc6d47d3adf8d15dc5d28d5d2bd12d6d7dda2d74dc6d
05 = 89db7df1d30df9d8adfddbfd6bdf8dc5decd66d13dbed5ad
06 = acdadd7ddebd3dd8fd21da4d71d4fdd2db4dbcde0d10d68d
07 = 72dfd3cdfad34d33de1d4fd46d4dd9cdcbd9cd77d12d32d
08 = dfddd6edefd9ddbddffde0dc3d11d58d8cdcdb5d73dafd
09 = 78da6d29de0d96dabd9fd2d71d9fdcdd9db7d3ad74dfcd
10 = 5ed72d97d35db8d71d1cdb6d79de7dbcd3ad19dfdd8dd56d
11 = 8bda1d14d86d29df2da7d7dda0d4dd5addad5fd2d1fdddd
12 = d4d1dd8cdd9d8fd0db2d4de9d80d9d98decdf8d42d7ed
13 = cbdccdbfdfcddbd7dd7ed77d39dc4d27d8dda1d25d4ed86d
14 = dad42decd1cd57df0d66df4ddadc1d72d29d42db3dfad7ed
15 = afd20d78d4fdbdd60d83d7fda3d30d5adcbdc3d3dd82de0d
16 = 72d64d45d84d18d55dabde6d2bd29dfad1fd55dc7de4d15d
17 = e6d95d5bdaadc8dc2d9fd33d7dd4d1edacd10deed67d76d
18 = 36d80d66d3fd81defd16d12d16ddedb7dbbd94d2ed7dd11d
19 = bad43dccdcdd18d52d3bd82d2ada8d0d8ed90da8dc2dd0d
20 = cfd38df3df7d42dc0ddbdd5d53de6d9bd39d6adfcd6ad20d
21 = b2dfddb8d99dccd8adc5d6edacde2d26d66dcdd7cdaed39d
22 = 47db7dbed82dd0d84d17d51d8fdd9db1de1d45d39d6ad38d
23 = ad14d0d7cd4ed7de9dd3d4cde6d20de6d57df1dcade0d
24 = c5d34dccd9fdc4d9d96d6ddcfd44d82d5cd19db5dd6dd6d
25 = 1fd53d57dceda3df6daed5fdb8de6d7fdc4d61dafde8d96d
26 = 18dedb6dcddc4d69d33d9bd5bdcad46df0d56d8fd1dc5d
27 = f1d4fdad1ed86db0d7dd2fd41df1df4d1fd25d36d46d54d
28 = 42d45d6dd2fd86d91d1dd78dd6dfdd2ed4fd4ad1d20d24d
29 = 6bdcad8dd97d19dc8d86d87d95d1ddf1d49dedd17d91d5dd
30 = ebd8cd7ed71de9d71d65dc2d33d65d57d11de5df3d7ad73d
31 = e0dbbdefd1ddfed38d27d22d0df4dd6dddacde3dcbd77d
32 = 3adebdc5dc4d4ad5addadf6d9ad8fde4d38decd50d90da5d
33 = 39d7d29d5dcdd80ddddedd5de3d5dda3d2ad57d4ed96d
34 = a8d7bdccda2d93d67d2ddaed34d38df1de2dbad95d3ad41d
35 = 6adf1d16daedc3d80db8dc8d3da2dfbde9dd7d84d72d83d
36 = f1dead89d1cd13dd6db9d8fde6d73d19d44d2bd11d90dcbd
37 = 81d72d4dd8cdbcdd5db6d43d86d2fd75dddd2dd85d67dbcd
38 = 5edf5d2ed7cd37d4cdded6bd3edd8d94dd6d4cdbad79d95d
39 = 20d57df0dbfd84df5d8bdecd92d6dd5ed22d4fd6d58debd

$E(D_1)$ = {[13 01 02 03 04 05 06 02 07 08 09 05 06 02 10 11 12 09 / 20 19 01 16 05 03 01 06 01 05 01 03 02 01 03 02 01 01]}

$E(D_2)$ = {[13 01 02 03 04 05 06 02 07 08 09 05 06 02 10 11 12 09 / 20 19 01 16 05 03 02 01 06 05 01 03 02 01 03 02 01 01]}

$E(D_3)$ = {[18 14 15 02 16 05 06 02 03 04 05 06 02 07 08 09 05 06 02 10 11 12 17 / 29 28 26 01 22 03 02 01 16 05 03 02 01 06 05 01 03 02 01 01 02 01 01 02 01]}

$E(D_4)$ = {[31 19 20 21 22 23 24 25 26 27 28 29 23 28 29 23 27 30 09 02 / 22 21 16 15 14 01 11 06 05 01 03 02 01 03 02 01 01 04 01 01]}

$E(D_5)$ = {[33 32 02 19 20 21 22 23 24 25 26 27 28 29 23 28 29 23 27 30 09 02 09 / 25 24 01 21 16 15 14 01 11 06 05 01 03 02 01 03 02 01 01 04 01 01 01]}

$E(D_6)$ = {[39 34 35 36 37 23 38 09 02 10 11 12 / 13 12 05 04 03 01 01 01 01 03 02 01]}

FIGURE 4 – Les empreintes des définitions de notre corpus jouet.

Matrice de dissimilarité :

$$\begin{pmatrix} 0 & 0 & 144 & \infty & \infty & \infty \\ 0 & 0 & 144 & \infty & \infty & \infty \\ 144 & 144 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 71 & \infty \\ \infty & \infty & \infty & 71 & 0 & \infty \\ \infty & \infty & \infty & 71 & \infty & 0 \end{pmatrix}$$

Évolution du partitionnement :

Étape 1	$\{1,2\},\{3\},\{4\},\{5\},\{6\}$	car 1 et 2 ont la même empreinte.
Étape 2	$\{1,2\},\{3\},\{\{4\},\{5\}\},\{6\}$	car 4 et 5 sont les plus proches.
Étape 3	$\{\{1,2\},\{3\}\},\{\{4\},\{5\}\},\{6\}$	car 3 et $\{1,2\}$ sont les plus proches.
Étape 4	$\{\{1,2\},\{3\}\},\{\{4\},\{5\}\},\{6\}$	car toutes les classes sont absolument dissimilaires.

FIGURE 5 – Trace du partitionnement des définitions du corpus jouet.

t	::=	x	*Variable*
	|	c	*Constante*
	|	$t(\overline{t})$	*Application*
	|	$\lambda\overline{x}.t$	*Abstraction*
	|	**let** b **in** t	*Définition locale*
	|	**let rec** $\overline{b}$ **in** t	*Définitions récursives*
	|	$\delta(\overline{t})$	*Appel d'une primitive*
	|	**switch** $t\,\{\,\overline{\gamma};\overline{\gamma};t\,\}$	*Branchement*
	|	**staticraise** $n(\overline{t})$	*Saut local*
	|	**statictry** t **with** $n(\overline{x}) \to t$	*Expression étiquetée*
	|	**try** t **with** $x \to t$	*Expression étiquetée*
	|	**if** t **then** t **else** t	*Branchement conditionnel*
	|	$t;t$	*Séquencement*
	|	**while** t **do** t **done**	*Boucle non bornée*
	|	**for** $x = t$ **to** t **do** t **done**	*Boucle bornée ascendante*
	|	**for** $x = t$ **downto** t **do** t **done**	*Boucle bornée descendante*
	|	$x := t$	*Affectation*
	|	$t\,\#\,t(\overline{t})$	*Appel de méthode*
b	::=	$x = t$	*Définition*
γ	::=	$n \to t$	*Branche*

FIGURE 6 – La syntaxe de LAMBDA avec $n \in \mathbb{N}, x \in \mathcal{V}, c \in \mathcal{C}, \delta \in \mathcal{P}$.

les primitives sont nécessairement appliquées. Le langage contient un fragment impératif permettant d'affecter des variables, d'itérer *via* des boucles `for` et `while`, et enfin de détourner le flot du contrôle *via* les différents mécanismes de lancement et rattrapage d'exceptions. L'appel de méthode est la construction qui nous rappelle qu'OCAML est un aussi un langage à objets. Pour finir, LAMBDA n'a pas d'analyse de motifs mais est muni d'un branchement n-aire introduit par le mot-clé `switch`. Par manque de place, nous ne donnons pas les règles de sémantique de ce langage et nous laissons au lecteur le soin de réfléchir à ces dernières. Par contre, pour se convaincre que l'on ne perd pas trop de structure en calculant la redondance sur des termes de LAMBDA et non des termes OCAML, il faut prendre le temps d'expliquer les différences entre ces deux langages.

Absence de types LAMBDA est un langage non typé. On n'y retrouve donc aucune déclaration de types. Cette absence limite donc d'emblée le champ d'application de ASAK : nous ne pouvons pas détecter de déclarations de type redondantes. Cependant, cette limitation a un avantage : lorsque l'on oublie les types, on se donne la possibilité de détecter plus de redondances entre des programmes de types distincts mais partageant la même structure. En revenant aux définitions des types qui interviennent dans deux programmes ayant la même structure, on peut espérer détecter indirectement des redondances entre les définitions de types. Un exemple d'une telle situation sera présentée et discutée dans la section 6.

Absence de modules et de classes Les constructions de seconde classe (comme les définitions de modules et de classes d'objets) ont disparu dans le programme traduit en LAMBDA. Cela limite notre capacité à détecter des modules similaires ou des classes similaires. Comme pour les définitions de type, nous pensons qu'en nous focalisant uniquement sur les aspects calculatoires, nous pouvons détecter *a posteriori* des définitions de modules ou de classes similaires parce qu'elles partagent des définitions similaires.

Absence d'analyse de motifs L'analyse de motifs d'OCaml a été compilée en Lambda sous la forme d'arbres de décision, exprimés à l'aide d'expressions conditionnelles et de branchements. Deux analyses de motifs distinctes syntaxiquement en OCaml peuvent être envoyées vers le même arbre de décision et donc le même code Lambda : c'est par exemple le cas de deux analyses à trois branches sur le type `color = Red | Black | White` car quelque soit l'ordre des branches de l'analyse[3], celles-ci vont être traduites vers des branchements à trois cas où l'ordre des cas correspond à l'ordre des constructeurs de données dans la définition du type. Il s'agit donc encore une fois d'une simplification favorable des termes sources puisqu'elle envoie des termes sources syntaxiquement distincts mais sémantiquement équivalents vers un unique terme Lambda.

α-renommage Le nom des variables est pris en compte lors de la prise d'empreintes. Naturellement, nous avons choisi de nous abstraire de ces noms en détectant la redondance à α-équivalence près. Il est donc important de s'assurer que les noms de variables liées (par exemple lors d'une définition de fonction) ne jouent aucun rôle. Pour cette raison, nous renommons tous les identificateurs des variables liées en des identificateurs canoniques qui codent leurs indices de De Bruijn.

Réduction des définitions locales inoffensives En OCaml, et contrairement à Coq, on ne peut pas facilement comparer les termes modulo l'évaluation : en effet, l'évaluation d'un terme peut diverger ou produire des effets de bord incontrôlés. Par contre, le compilateur sait détecter des définitions locales très simples dont l'expansion ne pose pas de problème. Ces définitions sont marquées par une annotation du type suivant :

```
type let_kind = Strict | Alias | StrictOpt | Variable
```

`Strict` signifie que la définition peut faire des effets de bord et ne doit pas être réduite (sauf s'il s'agit d'une variable ou d'une constante) ; `Alias` signifie que la définition est pure et peut donc être réduite ; `StrictOpt` signifie que la définition dépend de la mémoire et ne peut donc pas être réduite ; `Variable` signifie que la définition sera masquée dans la suite. Notre normalisation réduit les définitions `Strict` simples et les `Alias` ce qui nous permet d'égaliser des termes sources qui sont équivalents modulo le dépliage de ces définitions. [4]

3.2 Définition de la prise d'empreintes

Notre méthode de prise d'empreintes est une variante d'un calcul d'empreinte de la littérature [5]. L'empreinte d'un arbre de syntaxe doit témoigner de la structure de cet arbre ainsi que de l'ensemble des sous-arbres qui la constitue. Pour des raisons d'efficacité, nous réduisons chaque sous-arbre à un entier correspondant à son image par une fonction de hachage. Par ailleurs, plus un sous-arbre a un grand nombre de nœuds et plus sa contribution à l'arbre global est important : nous pondérons donc la clé de hachage par ce nombre de nœuds.

Definition 1
On appelle *f-glyphe d'un arbre de syntaxe t*, le couple $H_f(t)$ formé d'un entier 63 bits correspondant à son nombre de nœuds et d'un entier 128-bits correspondant à l'image de cet arbre à travers une fonction de hachage f donnée. Pour un glyphe g donné, on note son poids $w(g)$ et sa clé de hachage $h(g)$.

3. En supposant qu'il n'y a pas de clause `when` en jeu.
4. Notons que cette normalisation peut changer la complexité ce qui d'un certain point de vue ne préserve pas la sémantique intentionnelle du programme source. Dans nos expérimentations, ces considérations ne se sont pas montrées pertinentes cependant.

Definition 2
On appelle *f-empreinte d'un arbre de syntaxe t*, le multi-ensemble $E_f(t)$ formé des f-glyphes de ses sous-arbres et de lui-même.

Il reste maintenant à décider quelle fonction de hachage utiliser. Ne nous intéressant pas à des questions de sécurité, nous avons décidé d'utiliser intensivement la fonction de hachage cryptographique MD5. Pour chaque nœud de l'arbre, on calcule une clé de hachage qui s'appuie sur le nom de la construction utilisée et, pour procéder incrémentalement [8], sur les clés de hachage des sous-arbres prise dans l'ordre, de gauche à droite.

Dans un programme, la valeur d'un littéral est très souvent liée au contexte d'application. Du point de vue de la recherche de redondance, nous estimons que ces valeurs constituent une forme de bruit à ignorer. Pour que la valeur exacte des littéraux n'influence pas notre recherche de redondance, nous avons donc décidé de ne pas prendre en compte leurs valeurs dans le calcul de la clé de hachage.

Pour finir cette section, il faut remarquer qu'il n'est pas pertinent de garder tous les glyphes des sous-termes dans l'empreinte finale. En effet, nous avons vu que la fonction de partitionnement utilise le multi-ensemble des glyphes des sous-arbres pour identifier les codes "proches". Comme nous travaillons en s'abstrayant les constantes (c'est-à-dire que toutes les constantes ont le même glyphe), presque tous les arbres partagent ces glyphes. Nous ne pouvons donc pas conserver les empreintes de tous les sous-arbres. Une approche consiste à ne conserver que les empreintes des "gros" sous-arbres, afin d'éviter la pollution induite par les empreintes des feuilles. La définition même de "gros" étant sujette à cautions, nous proposons deux possibilités : (i) ne retenir que les empreintes des sous-arbres ayant au moins un certain pourcentage p de nœuds comparé au nombre de nœuds de l'arbre tout entier (utile quand on ne connaît pas a priori la taille des définitions étudiées) ; ou bien (ii) ne retenir que les empreintes des sous-arbres ayant un nombre n de nœuds (utile quand on ne s'intéresse qu'aux fonctions dont on connaît la taille). Il s'agit d'un paramètre global de Asak.

4 Partitionnement de corpus dirigé par la similarité

Partitionner un corpus vis-à-vis de l'égalité entre empreintes permet déjà de regrouper des programmes syntaxiquement distincts mais qui ont des structures calculatoires très proches, sinon égales. Nous souhaitons aller plus loin en regroupant des programmes qui se ressemblent même si leurs structures respectives diffèrent plus significativement. Considérons par exemple :

```
1 let id1 x = print_endline "debug"; x and id2 x = x
```

`id1` et `id2` n'ont pas le même glyphe mais on peut difficilement ignorer leur ressemblance !

Comme nous l'avons expliqué précédemment, nous utilisons un algorithme de partitionnement hiérarchique ascendant qui a besoin d'une notion de dissimilarité pour fonctionner.

Definition 3
Soient X et Y deux empreintes. La *dissimilarité* $d(X,Y)$ entre X et Y est une valeur de $\mathbb{N} \cup \{\infty\}$ définie comme suit :

$$d(X,Y) = \sum_{g \in X \Delta Y} w(g) \text{ ou } \infty \text{ si } X \cap Y = \emptyset$$

où $X \Delta Y$ est la différence symétrique entre deux multi-ensembles X et Y.

Cette définition n'est pas très standard car elle sépare de façon très brutale les programmes qui ne partagent aucun sous-terme. Ce choix garantit que deux programmes sans rapport ne pourront jamais apparaître dans la même classe. Ainsi, le résultat final sera composé d'une union

disjointe de classes infiniment distantes les unes des autres. Nous pensons que cette propriété améliore la lisibilité des résultats.

Notez que cette notion de dissimilarité n'est pas une distance. En effet, elle ne vérifie pas l'inégalité triangulaire. Par contre, c'est une fonction de séparation ($\forall X, Y,\ d(X,Y) = 0 \iff X = Y$) et symétrique ($\forall X, Y,\ d(X,Y) = d(Y,X)$). Cette propriété est suffisante pour appliquer l'algorithme de partitionnement.

L'algorithme procède par itération sur une liste de classes d'empreintes. On suppose que l'on sait fusionner deux classes d'empreintes et que l'on garde trace de ces fusions pour pouvoir produire un dendrogramme par classe.

```
(* Types pour les classes d'empreintes et les partitionnements. *)
type cluster and clustering
(* L'opération de fusion. *)
val merge : cluster -> cluster -> clustering -> clustering
(* Un partitionnement initial avec une empreinte par classe. *)
val initial : fingerprint list -> clustering
(* Le nombre de classes d'un partitionnement. *)
val size : clustering -> int
```

Il faut étendre la notion de dissimilarité aux paires de classes.

Definition 4
La dissimilarité $d(\alpha, \beta)$ entre deux classes d'empreintes α et β est :

$$d(\alpha, \beta) = \max_{X \in \alpha, Y \in \beta} d(X, Y)$$

Cette définition de la dissimilarité inter-classes est classique et donne lieu à un partitionnement dit "à liaison complète" (*complete linkage clustering*) [15].

Enfin, on suppose l'existence d'une fonction capable de déterminer les deux classes les plus proches dans le partitionnement courant :

```
type dissimilarity = Regular of int | Infinity
(* Renvoie les deux clusters les plus proches selon d, ainsi que leur distance *)
val get_closest_with_d : clustering -> (dissimilarity * (cluster * cluster))
```

L'algorithme de partitionnement hiérarchique est donné par le programme OCaml suivant :

```
(* Renvoie une liste de classes deux à deux infiniment dissimilaires. *)
let rec make_clustering xs =
  if size xs <= 1 then xs else match get_closest_with_d xs with
    | (Infinity, _) -> xs
    | (Regular p, (u, v)) -> make_clustering (merge u v xs)
let clustering fs = make_clustering @@ initial fs
```

Cette formulation de l'algorithme est malheureusement trop naïve pour être implémentée directement. En effet, en pire cas, la complexité de cet algorithme est cubique en fonction de la longueur de la liste `fs`, c'est-à-dire du nombre de définitions du corpus et dans nos cas d'usage, ce nombre de l'ordre de 10^6.

Heureusement, il est possible d'en fournir une version optimisée (restant de complexité cubique néanmoins). Cette optimisation s'appuie sur les quatre étapes suivants :

1. On commence par séparer les empreintes en deux ensembles : celles qui sont infiniment dissimilaires des autres et celles qui ne le sont pas. Cette phase permet d'éliminer les définitions qui ne partagent absolument rien avec les autres, et sont donc exclues de toute forme de redondance. Cette passe a un coût quadratique en le nombre D d'empreintes.

2. Les dissimilarités inter-empreintes peuvent être précalculées une fois pour toute pour un coût quadratique en le nombre D d'empreintes. Ce précalcul, qui est effectué en parallèle sur plusieurs cœurs, permet de répondre en temps logarithmique la valeur de $d(X, Y)$.
3. On calcule ensuite une surapproximation du partitionnement : à l'aide d'une structure de type *Union-Find*, on regroupe les empreintes qui ont une interdissimilarité finie. On sépare ainsi les empreintes qui ne pourront jamais être dans la même classe car elles sont infiniment dissimilaires deux-à-deux. Ce calcul a un coût en $O(D^2 \cdot logD \cdot \alpha(D))$.
4. Pour finir, l'algorithme de partitionnement naïf est exécuté en parallèle sur toutes les classes du partitionnement sur-approximé. Cette étape a un coût cubique en la taille de chaque partition, ce qui réduit significativement le temps d'exécution de l'algorithme car, sur nos exemples, la cardinalité des classes ne dépasse jamais 10^4.

5 Partitionnement de code étudiant

Motivation Une centaine d'étudiants de troisième année suivent le cours de programmation fonctionnelle. Ils ont deux heures de travaux pratiques par semaine produisant alors plusieurs centaines de réponses. Comment l'enseignant peut-il efficacement analyser ces réponses pour comprendre les difficultés rencontrées par ses étudiants ?

L'utilisation de LearnOCaml [3] est déjà d'une grande aide car la correction automatique des exercices permet de voir les pourcentages de réussite de chaque étudiant. Malheureusement cela ne suffit pas toujours car environ 80% des étudiants obtiennent tous les points ! Une analyse plus poussée vise à déterminer *comment* les étudiants ont répondu. Nous avons donc intégré Asak à LearnOCaml afin de classifier les codes des étudiants. En observant les représentants des classes obtenues et leur taille, l'enseignant peut se faire rapidement une idée de la façon dont les élève ont répondu et ainsi identifier les élèves qui ont fourni une réponse inattendue et qui nécessitent peut-être plus d'attention.

Approche utilisée Partant de l'hypothèse que les tests automatiques ont été bien écrits, nous classifions une première fois les codes selon la note qu'ils ont obtenue. En effet, nous ne voulons jamais identifier deux codes qui n'ont pas eu la même note, même s'ils sont similaires syntaxiquement. Cette première passe nous permet aussi d'avoir pour chaque classe une hypothèse, très forte, d'équivalence sémantique. Cette hypothèse nous permet de raffiner la fonction de calcul d'empreinte afin d'identifier encore plus de codes.

Amélioration de la fonction de calcul d'empreinte Comme nous l'avons vu précédemment, la clé de hachage d'un terme Lambda est déduite des clés de hachage de ses sous-termes *combinées dans l'ordre, de gauche à droite*. Dans cette situation, deux termes équivalents sémantiquement et qui ne diffèrent que par l'ordre de certains sous-termes ne sont pas envoyés vers la même empreinte. Pour résoudre ce problème, nous avons modifié notre fonction de prise d'empreintes pour qu'elle trie les empreintes des sous-termes avant de les combiner. Trier ou non les sous-empreintes est un paramètre global de Asak.

Nous avons déjà soulevé le problème du calcul des empreintes des feuilles de l'arbre en section 3.2. Dans ce contexte d'équivalence sémantique, nous pouvons supposer que dans deux arbres de même forme, les identifiants ne sont que des alias les uns des autres. En effet, s'il existait une différence sémantique entre deux identifiants, les deux codes n'auraient pas la même sémantique. Nous avons donc choisi d'associer la même empreinte à tous les identifiants.

Résultats Les exemples de la figure 4 sont tirés d'un corpus plus gros : celui des réponses des étudiants de troisième année à l'exercice "Implémentez la fonction `rev`". 154 réponses ont obtenu tous les points. Sur ce corpus, Asak produit 9 classes dont voici les cardinaux et une description succinte : 95 fois le premier élément est mis à la fin du reste de la liste renversée récursivement ; 41 fois la réponse introduit une fonction auxiliaire récursive terminale ; 5 fois la réponse utilise une fonction auxiliaire non-locale ; 4 fois le résultat de l'appel récursif est stocké dans une variable locale et ajoute le premier élément à la fin ; 3 fois la réponse utilise `List.rev` ; 3 fois la réponse s'appuie sur `List.fold_left` ; 2 fois la réponse s'appuie sur `List.fold_right` ; 1 fois la réponse utilise un `if-then-else` au lieu du filtrage par motifs.

L'enseignant peut déduire plusieurs choses de ce rapport. D'abord que la plupart des étudiants n'ont pas encore acquis le réflexe d'écrire des fonctions sur les listes dont la récursion est en position terminale. Il peut aussi constater qu'une minorité d'étudiants n'arrive pas encore à se détacher du paradigme impératif ou à utiliser l'analyse de motifs plutôt qu'une expression conditionnelle. Enfin, l'enseignant devra aussi revenir sur son code de correction automatique qui ne capture pas le cas de triche – pourtant grossier – où l'étudiant esquive l'exercice en réutilisant la fonction `List.rev`.

Pour des raisons de protection des données de nos étudiants, nous ne pouvons pas publier le jeu de données qui nous a servi à produire ces résultats. Des résultats similaires sont néanmoins reproductibles avec la version de développement de LearnOCaml [3]. Dans le mode "enseignant", il suffit de faire un clic du milieu sur le nom d'un exercice puis de spécifier la fonction à analyser pour obtenir le partitionnement des copies produit par Asak.

6 Détection de redondance dans les paquets OPAM

Approche utilisée Contrairement à l'environnement bien contrôlé des copies LearnOCaml, chaque paquet OPAM a un ensemble de dépendances et une logique de compilation potentiellement complexe. Pour ces paquets, la phase d'extraction des termes Lambda nécessite donc une configuration spécifique de la partie avant du compilateur. Il serait difficile d'automatiser cette configuration à partir du contenu des paquets. Nous avons préféré créer une version personnalisée du compilateur OCaml [2] installable dans *switch* OPAM. Le compilateur modifié instrumente la compilation standard par un mécanisme de normalisation et d'exportation des termes Lambda. Le dossier contenant le fichier source permet au compilateur de connaître le paquet OPAM en cours d'installation. Chaque terme collecté peut ainsi être étiqueté par le nom du paquet OPAM, par le nom du fichier source et par le chemin du module de la définition dont il est issu.

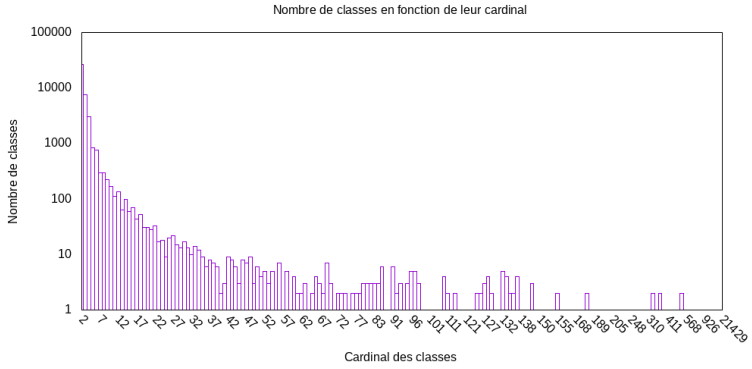
À l'aide d'un serveur de calcul [5], nous avons ensuite parallélisé les (tentatives d')installations de l'ensemble des paquets OPAM. Au bout de 20 heures, nous avons obtenu un corpus de travail dont nous rapportons maintenant l'analyse avec Asak.

Portée de l'analyse Nous avons réussi à compiler 1250 paquets sur les 2428 paquets que compte à ce jour le dépôt OPAM officiel. Tous les paquets n'ont pas pu être installés car certains ne compilent pas avec OCaml 4.08.1 ou sont en conflit avec OCaml 4.08.1 ; ou bien encore parce que certains paquets dépendent de bibliothèques C non-installées.

5. Une machine possédant 40 processeurs Intel(R) Xeon(R) CPU E5-4640 v2 cadencé à 2.20GHz équipé de 756Go de RAM.

Aspects quantitatifs Sur ces 1250 paquets, nous avons extrait 360152 définitions strictement différentes. On a retiré de ce décompte les définitions provenant de deux versions différentes d'un même paquet et qui produisent la même empreinte.

En ne conservant que les empreintes des sous-arbres de plus de 30 nœuds (limite calculable en temps raisonnable de notre implémentation), ces définitions ont été classées en 198841 classes, dont 40935 avec strictement plus d'un élément (c'est-à-dire que nous avons identifié au moins 40935 doublons). Le tout a été calculé en 40 minutes sur la machine décrite précédemment. Voici un diagramme à barre synthétisant la forme des classes obtenues (attention, l'échelle des ordonnées est logarithmique).



Aspects qualitatifs Les classes comprenant plus de 50 éléments sont majoritairement de deux types. Il s'agit d'une part de très petites fonctions (des variables globales ou des fonctions constantes). Comme Asak ne prend pas en compte la valeur des littéraux, ces dernières sont toutes regroupées. L'autre type de redondance est issu du code généré par des outils comme `Coq`, `Menhir`...

Cependant, certaines classes représentent de véritables possibilités de factorisation. Par exemple, nous trouvons une classe contenant 122 éléments comprenant toutes les définitions de la (célèbre) fonction `Option.map`, que l'on trouve sous pas moins de 32 noms différents. Cette classe capture aussi les fonctions similaires concernant les types isomorphes au type `option`. Par exemple, cette classe contient la fonction `map_evar_body` (provenant du module `Evd` de `Coq`) :

```
let map_evar_body f =
  function Evar_empty -> Evar_empty | Evar_defined d -> Evar_defined (f d)
```

Une synthèse complète de l'analyse des paquets OPAM par Asak sera l'objet de travaux futurs. La reproduction des résultats peut s'obtenir en suivant la procédure décrite en ligne à l'adresse https://github.com/nobrakal/asak/blob/master/utils/README.md.

7 Limitations

Nous programmons souvent en utilisant des constructions semblant similaires mais qui sont belles et bien des constructions sémantiquement différentes. Cependant Asak est par définition très sensible à la forme de l'arbre Lambda et ce "presque-sucre syntaxique" fait différer les arbres de codes pourtant très proches. Il en résulte qu'Asak n'est *pas* adapté à la détection de plagiat. Voici deux illustrations.

L'ordre des définitions L'ordre des définitions locales dans une fonction parait anodin mais il influe grandement le calcul d'empreinte puisque celui-ci est effectué récursivement. Il serait alors simple pour quelqu'un désirant fausser les résultats de réorganiser le code afin que sa sémantique reste préservée mais les arbres Lambda produit engendrent des empreintes différentes.

L'expansion des définitions Nous remplaçons souvent mentalement les variables par leur définition. Il s'agit cependant d'une opération complexe qui affecte souvent la sémantique du programme et que le compilateur se risque rarement à faire. Il en résulte que deux codes ayant exactement la même sémantique, l'un ayant une multitude de définitions factorisées et l'autre non engendrent des arbres très différents et donc des empreintes différentes.

8 Travaux connexes

La comparaison de code est un sujet très étudié. Différentes analyses détaillées des techniques existantes ont déjà été faites par Roy et al. [16] et Gautam et al. [9]. On peut grouper les différentes approches en quatre grandes familles qui utilisent chacune plus ou moins la sémantique du langage dans lequel le code est écrit en fonction de contraintes de performance ou de généricité.

Approche textuelle Il s'agit ici de comparer directement les chaînes de caractères composant le code, avec le plus souvent un pré-traitement visant à supprimer les espaces inutiles et les commentaires. Cette approche est la seule indépendante du langage (à l'exception de la phase de pré-traitement). Elle a été mise en œuvre dans des outils comme Duploc [6].

Approche lexicale D'autre outils choisissent de travailler sur la séquence de lexèmes correspondant au code. Ces outils deviennent donc dépendant d'un langage mais permettent de résoudre certains problèmes de l'approche purement textuelle (on est par exemple capable d'identifier deux codes égaux à α-renommage près). Cette méthode est implémentée dans des outils comme CC-Finder [12], CP-Miner [14] et DUP [17].

Approche syntaxique Une autre approche est d'utiliser directement l'arbre de syntaxe abstrait correspondant au code. Elle est donc dépendante du langage et permet de résoudre certains problèmes de l'approche lexicale. Cette méthode a été popularisée avec CloneDr [4] et utilisée plus récemment par Deckard [11].

Approche sémantique Enfin, on peut aussi utiliser les graphes de dépendance [7] du programme. Ces derniers permettent d'introduire beaucoup de sémantique dans la détection de clone mais la technique repose sur la recherche de sous-graphes identiques maximaux, problème qui est NP-complet. Des approximations en temps polynomial permettent néanmoins d'obtenir de bons résultats, comme l'a montré Krinke [13].

9 Conclusion et travaux futurs

Dans cet article, nous avons présenté l'approche suivie par Asak pour la détection de clones de programmes OCaml ainsi que des résultats préliminaires concernant l'application de cet outil

à LearnOCaml et à l'analyse du corpus de paquets OPAM. Pour pallier aux limitations que nous avons explicitées, nous allons continuer à améliorer la prise d'empreintes pour la rendre plus robuste à certaines transformations syntaxiques locales qui préservent la sémantique. Enfin, en intégrant Asak à un outil comme Merlin, nous allons proposer au programmeur un outil pour éviter d'introduire de la redondance dans les programmes OCaml.

Pour finir, les auteurs remercient la Fondation OCaml et ses sponsors pour avoir rendu possible le stage de licence d'Alexandre Moine sans lequel Asak n'aurait pas pu voir le jour.

Références

[1] Asak. https://github.com/nobrakal/asak.

[2] Compilateur OCaml pour la collecte de termes Lambda. https://github.com/nobrakal/ocaml branche 4.08.

[3] LearnOCaml. https://github.com/ocaml-sf/learn-ocaml.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998.

[5] Michel Chilowicz, Étienne Duris, and Gilles Roussel. Syntax tree fingerprinting : a foundation for source code similarity detection. Technical report, 2009.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 109–118, Aug 1999.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3) :319–349, July 1987.

[8] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19. ACM, 2006.

[9] Pratiksha Gautam and Hemraj Saini. Various code clone detection techniques and tools : A comprehensive survey. pages 655–667, 08 2016.

[10] Andrew Hunt and David Thomas. *The Pragmatic Programmer : From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder : a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7) :654–670, July 2002.

[13] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, Oct 2001.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner : finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3) :176–192, March 2006.

[15] Chandan K. Reddy and Bhanukiran Vinzamuri. A survey of partitional and hierarchical clustering algorithms. In Charu C. Aggarwal and Chandan K. Reddy, editors, *Data Clustering : Algorithms and Applications*, pages 87–110. CRC Press, 2013.

[16] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools : A qualitative approach. *Science of Computer Programming*, 74(7) :470 – 495, 2009.

[17] Brenda S. Baker. On finding duplication and near-duplication in large software systems. *Reverse Engineering - Working Conference Proceedings*, 08 2001.

Mesurer la hauteur d'un arbre

Jean-Christophe Filliâtre

CNRS

Résumé

Dans cet article, nous nous intéressons au problème du calcul de la hauteur d'un arbre. Le problème a l'air plutôt simple, à priori, puisqu'il suffit de suivre la définition mathématique avec une simple fonction récursive de quelques lignes. Néanmoins, une telle fonction peut facilement faire déborder la pile d'appels. Après avoir laissé le lecteur réfléchir à une solution, nous en discutons plusieurs, notamment au regard de ce qu'offre le langage de programmation. Ce problème illustre la difficulté qu'il peut y avoir à se passer de récursivité.

1 Le problème

Chacun sait que le soleil et le théorème de Thalès peuvent avantageusement être utilisés pour mesurer la hauteur d'un arbre, à l'instar de ce que Thalès lui-même fit pour mesurer la hauteur de la pyramide de Khéops. Le lecteur, cependant, aura compris qu'il s'agit plutôt ici d'écrire un programme informatique calculant la hauteur d'une structure de données arborescente.

Pour fixer le problème précisément, supposons qu'il s'agisse d'arbres binaires immuables. L'information contenue dans les nœuds ne nous intéresse pas ; on suppose cependant qu'elle ne stocke pas déjà la hauteur de chaque sous-arbre, sans quoi le problème serait trivial. Dans notre langage de programmation préféré, OCaml, un type pour de tels arbres peut être le suivant.

```
type tree = E | N of tree * tree
```

La hauteur est définie comme le nombre maximal de nœuds le long d'un chemin de la racine à une feuille (voir par exemple Knuth [5, Sec. 2.3]). De façon équivalente, on peut suivre la définition récursive du type `tree` pour définir ainsi la hauteur $h(t)$ d'un arbre t :

$$\begin{aligned} h(\texttt{E}) &= 0, \\ h(\texttt{N}(l,r)) &= 1 + \max(h(l), h(r)). \end{aligned}$$

La question qui nous intéresse ici est d'écrire une fonction `height`, de type `tree -> int`, recevant un arbre en argument et renvoyant sa hauteur. Il paraît évident de suivre la définition mathématique, c'est-à-dire d'écrire ce programme :

```
let rec height = function
  | E        -> 0
  | N (l, r) -> 1 + max (height l) (height r)
```

Cependant, cette solution a le défaut de faire déborder la pile d'appels pour un arbre dont la hauteur serait trop grande [1], ce qui est très facilement atteint avec des arbres qui seraient très linéaires, même avec peu de mémoire. Une telle situation n'est pas acceptable, notamment lorsqu'une fonction comme `height` fait partie d'une bibliothèque et que son auteur ne peut donc

1. Sur notre machine, cette valeur est de l'ordre de 275 000. Elle s'explique par une taille de pile de 8 Mo et des tableaux d'activation de 32 octets chacun (une adresse de retour, deux valeurs sauvegardées sur la pile et un alignement de chaque tableau sur 16 octets).

pas préjuger de son utilisation ni des conditions de son exécution au regard de la taille de la pile. Le problème que nous posons ici est donc de proposer une alternative à la fonction `height`, ayant le même type et la même spécification, mais s'exécutant en espace de pile constant.

Le lecteur est vivement invité à interrompre ici sa lecture
et à chercher une solution à ce problème.

Au delà du problème spécifique de la hauteur d'un arbre, c'est le problème plus général d'un éventuel débordement de pile par une fonction récursive qui nous intéresse. On entend souvent l'argument « il suffit d'utiliser une pile », justifié par le fait que c'est exactement ce que fait le compilateur. Mais lorsqu'il s'agit de le faire en pratique, c'est rapidement difficile. Si le lecteur a pris le temps de chercher une solution à notre problème, il doit maintenant en être convaincu.

Si nous avons choisi la hauteur d'un arbre plutôt que sa taille, c'est justement pour qu'il ne soit pas immédiat de simplement déposer des sous-arbres sur une pile, en accumulant la taille dans une variable globale. Le fait de devoir calculer un maximum *après* le calcul de la hauteur des deux sous-arbres est justement ce qui rend le problème intéressant. On peut analyser ainsi les raisons de cette difficulté : lorsque le compilateur utilise la pile d'appels pour compiler notre fonction récursive `height`, il y dépose des données (un sous-arbre dont la hauteur reste à calculer, une hauteur déjà calculée) mais également du contrôle, sous la forme d'*adresses de retour*. Lorsqu'on utilise une structure de pile explicite, il est facile d'y déposer des données mais moins évident d'y déposer du contrôle.

Dans cet article, nous présentons différentes solutions à notre problème. Certaines sont spécifiques au calcul de la hauteur d'un arbre (Section 2) et d'autres s'appliquent en revanche à toute fonction récursive ou toute structure arborescente (Section 3). Nous en comparons les performances (Section 4) avant de considérer quelques variantes du problème (Section 5) et de conclure. Tous les programmes décrits dans cet article sont accessibles sur la page https://www.lri.fr/~filliatr/hauteur/.

2 Des solutions ad hoc

Dans cette section, nous présentons deux premières solutions, spécifiques au calcul de la hauteur d'un arbre. Nous verrons dans la section 4 qu'elles sont particulièrement efficaces.

Parcours en largeur. Certains lecteurs auront sûrement imaginé utiliser un parcours en largeur et c'est là une excellente solution, concise et efficace. Un tel parcours en largeur peut être réalisé par une fonction qui manipule un entier `m` et deux listes, une liste `curr` contenant des nœuds de l'arbre situés à la profondeur `m` et une liste `next` des nœuds situés à la profondeur `m`+1. Lorsque la première liste vient à se vider, on incrémente `m` et on échange les deux listes. Lorsque les deux listes sont vides, la valeur de `m` est renvoyée.

```
let rec hbfsaux m next = function
  | []               -> if next=[] then m else hbfsaux (m+1) [] next
  | E       :: curr -> hbfsaux m next         curr
  | N (l, r) :: curr -> hbfsaux m (l::r::next) curr
```

Il n'y a pas lieu d'utiliser de file dans ce parcours en largeur, car l'ordre du parcours des nœuds d'une même profondeur ne nous intéresse pas. Il s'avère que l'on réalise ici un *boustrophédon* (une alternance de parcours de la gauche vers la droite puis de la droite vers la gauche). Il ne reste plus qu'à démarrer le parcours en largeur avec une liste `curr` contenant l'arbre tout entier, une liste `next` qui est vide et un accumulateur valant 0.

```
let hbfs t = hbfsaux 0 [] [t]
```

Il est important de noter ici que les trois appels récursifs à `hbfsaux` sont *terminaux*. Le compilateur OCaml optimisant de tels appels, en les remplaçant par des sauts [2], cette solution s'exécute bien en espace constant. Bien entendu, il serait très facile de réécrire la fonction `hbfsaux` avec une boucle `while`, même si elle serait sans doute rendue un peu moins élégante par la présence de références.

Comme on le voit, les arbres vides `E` sont ajoutés dans la liste `next` comme les autres, puis ignorés par le deuxième cas du filtrage dans `hbfsaux`. On peut modifier le code pour ne jamais ajouter d'arbre `E` dans les listes manipulées. On perd en lisibilité mais le programme obtenu est 40% plus rapide.

Avec une pile. Une autre solution ad hoc consiste à utiliser une pile contenant des sous-arbres, où chaque sous-arbre est accompagné de sa profondeur dans l'arbre original. On a donc une pile contenant des paires. Outre cette pile, le code maintient dans un accumulateur `m` le maximum des profondeurs rencontrées.

```
let rec hstackaux m = function
  | []                 -> m
  | (n, E)        :: s -> hstackaux (max m n) s
  | (n, N (l, r)) :: s -> hstackaux m         ((n+1,l) :: (n+1,r) :: s)
```

Comme le lecteur le constate, il s'agit maintenant d'un parcours en profondeur. Là encore, les deux appels récursifs sont terminaux et on a donc bien une solution en espace constant. Pour calculer la hauteur d'un arbre t, il suffit de démarrer avec la paire $(0, t)$.

```
let hstack t = hstackaux 0 [0, t]
```

Bien que plus concise encore que la précédente, cette solution est bien moins efficace (plus de deux fois plus lente). L'une des raisons est une sollicitation plus importante du GC, car on alloue deux fois plus de blocs (deux blocs par nœud maintenant). On peut y remédier avec un type ad hoc de listes de paires, comme celui-ci :

```
type stack = Nil | Cons of int * tree * stack
```

On obtient alors une version 30% plus rapide. Elle reste cependant moins performante que `hbfs`, notamment parce qu'on y fait beaucoup plus d'arithmétique : deux additions par nœud et des appels à `max`, là où `hbfs` ne fait qu'une addition par changement de niveau. En expansant la fonction `max`, et surtout en évitant d'empiler des arbres vides, comme nous l'avions fait pour `hbfs`, on obtient au final un programme très efficace.

3 Des solutions génériques

Dans cette section, nous présentons des solutions plus générales, qui fonctionnent pour toute fonction récursive ou pour le parcours de toute structure récursive.

2. On peut le vérifier en examinant le code assembleur produit par `ocamlopt`, avec l'option `-S` de ce dernier.

Passage de continuations. Une solution élégante consiste à adopter un style *par continuation* (en anglais CPS, pour *Continuation-Passing Style* [8]). L'idée est de généraliser le problème, en ne calculant pas la hauteur $h(t)$ de l'arbre t, mais $k(h(t))$ pour une fonction k passée en argument, appelée « continuation ». La solution s'en déduira au final en prenant pour k la fonction identité. Mais le fait d'avoir généralisé le problème va le rendre plus simple, comme souvent en mathématiques. D'une part, il est à peine plus complexe d'écrire une telle fonction. La voici :

```
let rec hcpsaux t k = match t with
  | E        -> k 0
  | N (l, r) -> hcpsaux l (fun hl ->
                hcpsaux r (fun hr -> k (1 + max hl hr)))
```

D'autre part, et c'est là tout l'intérêt, on note que *tous* les appels faits dans cette fonction sont maintenant des appels terminaux, aussi bien les deux appels récursifs à `hcspaux` que les deux appels à `k`. Dès lors, cette fonction s'exécutera bien en espace de pile constant, tout comme la fonction `hcps` qu'on en déduit.

```
let hcps t = hcpsaux t (fun h -> h)
```

Dans cette solution, toute l'information nécessaire au calcul est contenue dans les *clôtures* allouées sur le tas. Ainsi, la clôture correspondant à `fun hl ->` capture les valeurs de `r` et `k` et la clôture correspondant à `fun hr ->` capture les valeurs de `hl` et `k`. En particulier, comme chaque clôture capture la valeur d'une autre continuation `k`, les clôtures forment une liste chaînée. Cette liste se termine avec la continuation identité (`fun h -> h`) qui a été passée en argument initialement.

Bien entendu, une telle solution suppose deux choses : que notre langage propose des fonctions de première classe et que les appels terminaux soient correctement optimisés. Dans certains langages, on ne dispose tout simplement pas de fonction de première classe. Un exemple est le langage C. Dans d'autres langages, on dispose de fonctions de première classe mais les appels terminaux ne sont pas optimisés ou ne le sont pas tous. C'est le cas des langages Java ou Kotlin[3], pour n'en citer que deux.

Lorsque le langage ne propose pas de fonction de première classe, ou que son compilateur n'optimise pas les appels terminaux, le programmeur peut néanmoins utiliser la solution du passage de continuations. Il lui suffit de *défonctionnaliser* le programme [7, 2], c'est-à-dire de remplacer les clôtures par un type ad hoc. Ici, ce type fait la somme des trois continuations différentes (les trois constructions `fun` du code ci-dessus).

```
type cont = Kid | Kleft of tree * cont | Kright of int * cont
```

On reconnaît bien là une structure de liste chaînée, comme identifiée plus haut. Il faut maintenant écrire *deux* fonctions, l'une qui est la version défonctionnalisée de `hcpsaux` et une autre pour appliquer une continuation à un argument.

3. En Kotlin, seuls les appels terminaux *récursifs* sont optimisés, à la demande explicite du programmeur, mais un appel récursif fait à l'intérieur d'une clôture ne le sera pas. Dans le cas de notre fonction `hcpsaux`, cela veut dire qu'un seul des quatre appels sera optimisé, à savoir le premier appel récursif à `hcpsaux`.

```
let rec hdefunaux t k = match t with
  | E        -> hdefuncont k 0
  | N (l, r) -> hdefunaux l (Kleft (r, k))
and hdefuncont k v = match k with
  | Kid            -> v
  | Kleft (r, k)   -> hdefunaux r (Kright (v, k))
  | Kright (hl, k) -> hdefuncont k (1 + max hl v)
```

Comme on le voit, il s'agit de deux fonctions mutuellement récursives, où tous les appels sont toujours des appels terminaux. Il ne reste plus qu'à appeler `hdefunaux` avec la continuation `Kid`.

```
let hdefun t = hdefunaux t Kid
```

Si le compilateur n'optimise pas correctement l'appel terminal, on peut modifier encore une fois le programme pour remplacer ces deux fonctions mutuellement récursives par une boucle `while`. Le code en ligne qui accompagne cet article contient également cette version.

Le zipper. Une autre solution générique, s'appliquant au parcours d'une structure récursive quelconque, consiste à utiliser un *zipper* [4]. Il s'agit là d'une structure de données permettant de désigner un sous-terme d'une structure récursive et d'effectuer des déplacements locaux. Dans le cas d'un arbre binaire, qui nous intéresse ici, cela veut dire désigner un sous-arbre et se déplacer en descendant dans son sous-arbre gauche ou son sous-arbre droit ou en remontant au nœud parent.

Le *zipper* implémente un tel curseur comme une paire (p, t), où p est le chemin entre la racine et la position considérée et t le sous-arbre se trouvant à cette position. L'idée clé du *zipper* consiste à représenter le chemin depuis la position considérée jusqu'à la racine, plutôt que l'inverse. Ainsi, les déplacements s'opèrent en tête de liste, en temps constant. Voici un type `path` pour de tels chemins :

```
type path = Top | Left of path * tree | Right of tree * path
```

La constant `Top` représente la racine de l'arbre ; une valeur `Left`(p, t) (resp `Right`(t, p)) représente un sous-arbre gauche (resp. droit) dont le parent est désigné par le chemin p et dont le sous-arbre droit (resp. gauche) est t. On peut alors facilement définir de petites fonctions `left`, `right` et `up` pour se déplacer dans l'arbre (voir l'article de Huet [4]) et en déduire facilement un parcours de l'arbre sous la forme d'une boucle. Le code est donné en ligne. Comme pour les solutions précédentes `hcps` et `hdefun`, on a remplacé de l'espace utilisé sur la pile par de l'espace utilisé sur le tas, ici avec des valeurs du type `path` [4].

4 Performances

On donne ici une mesure des performances de ces diverses solutions. Pour chaque fonction, on mesure le temps total passé dans le calcul de la hauteur

— de tous les peignes à gauche de hauteur n avec $0 \leq n \leq 10\,000$;
— de tous les peignes à droite de hauteur n avec $0 \leq n \leq 10\,000$;
— de tous les arbres binaires parfaits de hauteur n avec $0 \leq n \leq 20$;
— de cent arbres construits aléatoirement avec $2000 \times n$ nœuds avec $0 \leq n < 100$.

4. Un lien peut être fait entre la version défonctionnalisée de la solution CPS et cette solution utilisant un *zipper* ; ceci est notamment discuté par Danvy [1].

La construction des arbres n'est pas inclue dans cette mesure. Les arbres aléatoires sont les mêmes pour chaque mesure. La mesure se fait en répétant cinq fois le calcul, puis en éliminant la valeur la plus petite et la valeur la plus grande, pour enfin faire la moyenne des trois valeurs restantes. Les mesures ont été faites avec OCaml version 4.07.1 sur un unique processeur Intel Core i7 1.90 GHz. Les temps sont donnés en secondes.

version	temps
`height`	1,28

version	temps
`hbfs`	0,94
`hbfs_opt`	0,58
`hstack`	2,32
`hstack_opt`	0,52

version	temps
`hcps`	2,55
`hdefun`	1,80
`hzipper`	3,75

À titre de comparaison, on a inclus la fonction `height` dans le tableau de gauche, même si elle n'est pas acceptable en pratique. Les versions appelées `_opt` sont celles qui évitent d'ajouter des arbres vides dans les listes. Les amoureux de la programmation fonctionnelle seront sûrement un peu déçus de voir que la version CPS est relativement peu efficace, même une fois défonctionnalisée.

5 Variantes

Arbre n-aire. Une première variante consiste à généraliser au calcul de la hauteur d'un arbre n-aire. Un type pour de tels arbres peut être le suivant :

```
type tree = N of tree list
```

On note qu'il n'y a plus d'arbre vide ; tout arbre contient maintenant au moins un nœud. Les programmes s'adaptent néanmoins très facilement. Ils sont donnés en ligne.

Arbre mutable. Une autre variante consiste à considérer le cas d'un arbre binaire *mutable*. Dans ce cas, on peut effectuer un parcours infixe de l'arbre *en espace constant*, en modifiant puis restaurant la structure de l'arbre pendant le parcours. Un tel algorithme est dû à Morris [6]. La figure 1 contient un programme C réalisant un tel parcours, modifié pour calculer et renvoyer la hauteur. Les explications sont dans les commentaires. Quoi de mieux pour célébrer le quarantième anniversaire de cet article merveilleux ! À titre de comparaison, les performances d'un tel programme — écrit en OCaml pour comparer des choses comparables — sont de $1,55\ s$ sur les tests décrits section 4.

On pourrait également utiliser l'algorithme de Schorr-Waite [3], un algorithme général pour parcourir un graphe en inversant les pointeurs puis en les restaurant. Mais l'algorithme de Schorr-Waite nécessite de pouvoir stocker un peu d'information dans chaque nœud, en l'occurrence un bit d'information par nœud dans le cas d'un arbre binaire, afin de retenir si on était descendu dans le sous-arbre gauche ou droit.

6 Conclusion

Dans cet article, nous avons exploré la difficulté qu'il peut y avoir à transformer un programme susceptible de faire déborder la pile d'appel en un programme s'exécutant en espace de pile constant. Nous avons pris l'exemple du calcul de la hauteur d'un arbre, car il contient toute la difficulté du problème tout en étant concis. Nous avons proposé de nombreuses solutions,

```c
typedef struct T* tree;
struct T { tree left, right; };

int height(tree t) {
  int d = 0, h = 0;
  while (t != NULL) {
    if (t->left == NULL) { // pas de sous-arbre gauche, on descend à droite
      t = t->right;
      h = max(h, ++d);
    } else {
      tree p = t->left;    // sinon, on cherche le prédécesseur p de t
      int delta = 1;
      while (p->right != NULL && p->right != t) {
        p = p->right;
        delta++;
      }
      if (p->right == NULL) { // on le trouve pour la première fois
        p->right = t;         // on le fait pointer sur t
        t = t->left;          // et on descend à gauche
        h = max(h, ++d);
      } else {                // on le trouve pour la seconde fois
        p->right = NULL;      // on restaure l'arbre
        t = t->right;         // et on descend à droite
        d -= delta;
      }
    }
  }
  return h;
}
```

FIGURE 1 – Calcul de la hauteur avec l'algorithme de Morris (en C).

certaines ad hoc, utilisant des parcours en largeur ou en profondeur, et d'autres génériques, utilisant une transformation CPS ou un *zipper*.

Bien évidemment, une solution plus simple encore consisterait à stocker la hauteur dans l'arbre, en la calculant (en temps constant) chaque fois qu'un nœud est construit. Et c'est évidemment ce qu'il faut faire si la hauteur doit être calculée souvent, par exemple lorsque l'on implémente des AVL.

Nous avons utilisé le langage OCaml pour la présentation des diverses solutions mais toute cette discussion n'est en rien spécifique à OCaml. Le problème du débordement de pile se présente en effet dans (presque) tous les langages de programmation. Les solutions que nous avons proposées s'adaptent plus ou moins facilement dans un autre langage, notamment selon que ce langage propose des fonctions comme valeurs de première classe et que son compilateur optimise les appels terminaux.

Enfin, il est important de faire remarquer que toutes les solutions que nous avons présentées dans cet article utilisent un espace $O(N)$ dans le pire des cas, où N est la taille de l'arbre. Si l'arbre occupe une grande partie de la mémoire, on n'a pas forcément le loisir d'utiliser un

espace aussi grand pour en calculer la hauteur. Le code en ligne accompagnant cet article inclut un programme, dû à Martin Clochard, qui calcule la hauteur en espace $\log(N)$. Son temps de calcul, en revanche, est prohibitif.

Remerciements. Je remercie toutes les personnes, collègues ou étudiants, qui se sont prêtées au jeu lorsque j'ai testé sur elles ce problème.

Références

[1] Olivier Danvy. Defunctionalized interpreters for programming languages. *SIGPLAN Not.*, 43(9) :131–142, September 2008.

[2] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174. ACM Press, 2001.

[3] David Gries. The schorr-waite graph marking algorithm. *Acta Inf.*, 11 :223–232, 1979.

[4] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, September 1997.

[5] Donald E. Knuth. *The Art of Computer Programming, volumes 1–4A*. Addison Wesley Professional, 1997.

[6] Joseph M. Morris. Traversing binary trees simply and cheaply. *Inf. Process. Lett.*, 9(5) :197–200, 1979.

[7] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[8] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3) :233–247, Nov 1993.

Et TInA RUSTInA le lien vers l'assembleur
– étude préliminaire –

Frédéric Recoules[1], Sébastien Bardin[1], Richard Bonichon[2],
Laurent Mounier[3] et Marie-Laure Potet[3]

[1] CEA LIST, Laboratoire de Sûreté et Sécurité du Logiciel, Paris-Saclay, France
prenom.nom@cea.fr
[2] Tweag I/O, Paris, France
prenom.nom@tweag.io
[3] Univ. Grenoble Alpes. VERIMAG, Grenoble, France
prenom.nom@univ-grenoble-alpes.fr

Résumé

Le développement en C de logiciels non-critiques utilise régulièrement l'insertion d'assembleur « en ligne », que ce soit pour optimiser certaines opérations ou pour accéder à des primitives systèmes autrement inaccessibles. Celui-ci demande l'écriture de spécifications sous forme de contraintes pour faire le lien entre le langage hôte (C) et le langage assembleur embarqué. Ces spécifications sont ensuite utilisées par le compilateur afin d'insérer aveuglément l'assembleur dans le code émis – la pratique étant d'accorder toute confiance aux dires du programmeur. Pour éviter les erreurs issues de spécifications mal formées, nous proposons RUSTInA, un outil qui permet de vérifier que les spécifications assembleurs correspondent bien à l'implémentation des blocs qu'elles décrivent, ou, dans le cas contraire diagnostiquer ou corriger le problème.

1 Introduction

Contexte. Nous considérons ici la programmation de code « mixte », combinant assembleur embarqué et code C/C++. Cette fonctionnalité, présente dans les compilateurs GCC, clang et icc (Intel), permet d'intégrer des instructions assembleur au sein d'un programme C/C++. Elle est utilisée en général pour des raisons d'efficacité ou d'accès à des fonctionnalités bas niveau qui ne peuvent être déclenchées depuis le langage hôte. L'usage d'assembleur en ligne est plus fréquent que l'on n'imagine : ainsi **11%** des paquets Debian 8.11 écrits en C/C++ utilisent de l'assembleur en ligne, directement ou à travers des bibliothèques, avec des blocs allant jusqu'à 500 instructions, et **28%** des projets C les plus populaires sur Github en contiennent, d'après Rigger et coll. [8]. L'assembleur en ligne est ainsi fréquemment utilisé dans des domaines comme la cryptographie, le multimédia ou les pilotes matériel.

Problème. L'écriture d'assembleur en ligne comporte par nature un risque important : le compilateur fait entièrement confiance à l'interface du bloc assembleur, et notamment aux spécifications qu'elle comporte. Ceci peut donc entraîner des problèmes, notamment lorsque les spécifications reflètent incorrectement la sémantique du bloc concerné, ou réduire les opportunités d'optimisations – et avec elle les performances pourtant visées – si les contraintes se révèlent trop fortes.

Objectif. Nous souhaitons concevoir et développer une technique automatique, générique et pratique pour s'assurer de la conformité des instructions assembleurs avec les spécifications de l'interface qui leur correspond. De manière générale, nous entendons ainsi :

— vérifier la correction et la complétude de l'interface par rapport au code qu'elle décrit ;

— proposer un correctif pour que ces propriétés soient respectées si ce n'est pas le cas.

Fehnker et coll. [4] ont abordé ce problème de vérification des spécifications de l'interface. Cependant, leur technique ne couvre pas l'ensemble des propriétés que nous jugeons indispensables. Par ailleurs, la technique requiert un analyseur syntaxique dédié à l'assembleur embarqué pour chaque architecture, mais aussi, pour chaque dialecte assembleur. Testée sur ARM, elle souffre ici d'un manque de généricité et pourrait se révéler fragile sur une architecture de type *CISC*, plus complexe.

Contribution. Nous proposons RUSTINA, Rafistolage Utile de Spécifications par TINA, un outil de révision d'interface pour extraire la sémantique du bloc assembleur, vérifier sa conformité et proposer une rustine corrective si la correction n'est pas assurée, mais aussi, si le bloc est sur-contraint. Nos contributions se situent à 2 niveaux :

— l'identification des propriétés de conformité attendues entre une interface et son code assembleur (Sec. 3),

— un algorithme de vérification desdites propriétés et de génération automatique de correctifs, implémenté au-dessus de TINA[6, 7] (Sec. 4) et testé sur des projets d'envergure (Sec. 5).

Discussion. Notre technique trouve et corrige des bugs de *non-conformité* au niveau des blocs assembleur embarqué. En fonction du contexte, certains produiront alors des erreurs à l'exécution (Sec. 2), là où d'autres resteront fortuitement silencieux (Sec. 5).

2 Contexte et motivation

```
1563  # ifdef __PIC__

1566  __strcspn_g (const char *__s, const char *__reject)
1567  {
1568    register unsigned long int __d0, __d1, __d2;
1569    register const char *__res;
1570    __asm__ __volatile__
1571      ("pushl        %%ebx\n\t"
1572       "movl        %4,%%edi\n\t"
1573       "cld\n\t"
1574       "repne; scasb\n\t"
1575       "notl         %%ecx\n\t"
1576       "leal         -1(%%ecx),%%ebx\n"
1577       "1:\n\t"
1578       "lodsb\n\t"
1579       "testb         %%al,%%al\n\t"
1580       "je         2f\n\t"
1581       "movl         %4,%%edi\n\t"
1582       "movl         %%ebx,%%ecx\n\t"
1583       "repne; scasb\n\t"
1584       "jne         1b\n"
1585       "2:\n\t"
1586       "popl         %%ebx"
1587       : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
1588       : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
1589       : "memory", "cc");
1590    return (__res - 1) - __s;
1591  }

1618  # endif
```

FIGURE 1 – Code assembleur en ligne issu de libc6-dev

Exemple. L'extrait de code assembleur en ligne (x86) en Fig. 1 provient du fichier `/bits/string.h` du paquet libc6-dev (2.19-18) et nous servira d'exemple. Cette fonction calcule la longueur de la sous-chaîne de `__s` qui ne contient aucun caractère de `__reject`. Regardons comment l'assembleur est

lié au contexte C : les lignes 1587 à 1589 déclarent un contrat composé de 3 listes (séparées par `':'`) : les sorties, les entrées et les effets bord (appelés *clobbers*) du bloc. Les entrées et sorties associent des variables ou des expressions du C à des registres ou emplacements mémoire de l'assembleur. Elles sont numérotées dans l'ordre d'apparition, à partir de `0`, ce qui permet d'y faire référence dans les mnémoniques – le compilateur remplacera lors de l'émission du code les `%0-9` par l'emplacement qu'il aura choisi en respectant les contraintes. Ces dernières se composent de classes génériques (`'m'` mémoire, `'r'` registre général, etc.), de classes spécifiques (`'a' eax`, `'A' edx::eax`, etc.) et de modificateurs (`'='` écriture, `'+'` lecture/écriture, etc.). Il est possible de partager explicitement un emplacement entre une sortie et une entrée, en attribuant le numéro de la sortie à l'entrée (`"0"`). Le compilateur peut également le faire de son propre chef – il cherchera à minimiser le nombre de registres alloués et, pour ce faire, travaille avec l'hypothèse que les entrées sont « consommées » avant que les sorties ne soient « produites ». Dans cette optique, il est alors valide de partager le même registre. Toutefois, si la structure du code ne respecte pas cette hypothèse, le modificateur `'&'`, « écriture en avance », permet de forcer le compilateur à choisir un registre de sortie distinct de toutes les entrées.

En détail, voici comment interpréter le contrat en Fig. 1 :

%0. `"=S" (__res)` indique que le registre `esi` retournera la nouvelle valeur de `__res` et `"0" (__s)` qu'il doit contenir la valeur de `__s` à l'entrée du bloc.

%1-2. `"=&a" (__d0)` (respectivement `"=&c" (__d1)`) indique que le registre `eax` (respectivement `ecx`) sera écrasé – `__d0` (respectivement `__d1`) n'étant pas utile au delà du bloc, il s'agit d'un registre intermédiaire (scratch register) – et `"1" (0)` (respectivement `"2" (0xffffffff)`) qu'il doit contenir la valeur `0` (respectivement `0xffffffff`) à l'entrée du bloc.

%3. `"=&D" (__d2)` indique, comme précédemment, que le registre `edi` est un registre intermédiaire. Sa valeur à l'entrée du bloc n'est pas définie.

%4. `"r" (__reject)` indique qu'un registre en lecture seule – au choix du compilateur – doit contenir la valeur de `__reject`.

⚠. `"memory"` et `"cc"` indiquent que le bloc peut accéder et modifier respectivement tout emplacement mémoire accessible depuis l'environnement et le registre de condition.

Nous pouvons déduire de ces informations quelles sont les entrées qui partagent ou ne peuvent partager un registre avec une sortie ; ces données sont illustrées dans la Fig. 2. Le compilateur a ici peu de marge de manœuvre, l'interface est très contrainte car les mnémoniques sont très spécialisées et utilisent implicitement des registres particuliers. Le seul choix qu'il lui reste à faire est d'*associer* un registre à l'entrée `%4` (lignes 1572 et 1581), en sachant qu'il ne peut choisir `eax`, `ecx` ou `edi` à cause du modificateur `'&'`. De plus, tant que les valeurs de `__s` et `__reject` ne sont pas démontrées égales, `%4` se doit d'être distinct d'`esi`. Ce choix se résume donc aux registres `edx` et `ebx` – il semble d'apparence simple mais nous allons expliquer maintenant pourquoi il peut être fatal.

Le fond du problème. En effet, si `ebx` n'apparaît pas dans le contrat, il est bien utilisé dans le code assembleur (écrit ligne 1576, lu ligne 1582), à l'insu des compilateurs, puisque ceux-ci n'inspectent pas les mnémoniques. Ainsi, si ce registre est choisi, la valeur de l'entrée (le pointeur `__reject`) sera écrasée à la ligne 1576 par le calcul d'`ecx` (la longueur de la chaîne pointée par `__reject`). Le problème est alors d'autant plus important que, s'agissant d'un pointeur, la lecture (ligne 1581) et le déréférencement (ligne 1583) de la mauvaise valeur provoquera une *erreur de segmentation*. Cette erreur est ainsi observable à l'exécution lorsque le code est compilé avec GCC 5.4 et les options `-m32 -O3 -fpic`.

Remarque. La forme du contrat observée ici est probablement due au fait que le compilateur refusait, au moins jusqu'à GCC 4.8, la déclaration d'`ebx` en *clobber* si l'option `-fpic` était active : il était ainsi utilisé de façon détournée, en prenant soin de le sauvegarder (`push` ligne 1571) et de le restaurer (`pop` ligne 1586). Toujours est-il que l'hypothèse *contextuelle* – `ebx` ne sera pas choisi – est particulièrement

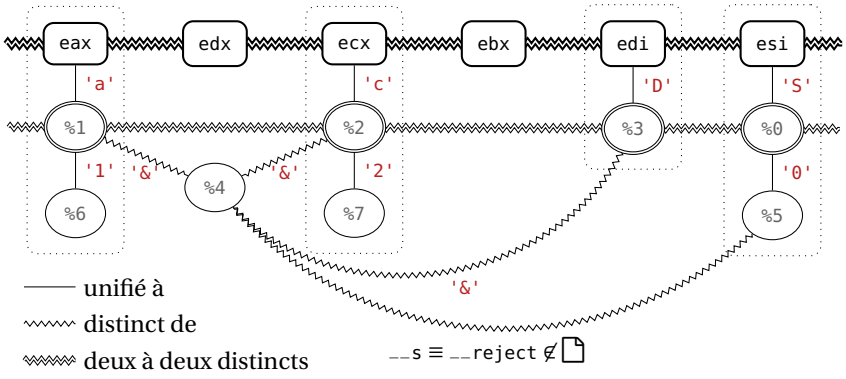


FIGURE 2 – Diagramme de juxtaposition

fragile avec l'évolution des compilateurs et la réutilisation de code. Aussi, afin de ne plus souffrir de pareille ambiguïté, *il faut associer explicitement* `%4` au registre `edx` à l'aide de `'d'` au lieu de `'r'`, ce que l'on peut observer dans des versions plus récentes de libc6-dev. Cet extrait montre donc que de subtiles erreurs peuvent aussi se glisser dans le code de programmeurs expérimentés.

Conclusion. À la complexité d'écrire de l'assembleur s'ajoute ainsi celle du contrat le liant au C. C'est pourquoi nous proposons en Sec. 3 de définir plus en détail les propriétés attendues de l'assembleur embarqué avant de présenter les caractéristiques techniques de notre outil RUSTINA en Sec. 4. Les résultats préliminaires de notre prototype sont eux discutés en Sec. 5.

3 Détails des propriétés recherchées

Nous définissons un ensemble de 5 propriétés que se doit de respecter une interface vis-à-vis de ses mnémoniques assembleur pour être qualifiée de ***BUENO***. Les 4 premières ***BUEN*** sont des propriétés de correction : elles sont nécessaires au bon fonctionnement du code. Elles sont menacées par les sous-spécifications qui, à l'instar des comportements indéfinis, provoquent de subtils bogues selon le bon vouloir du compilateur. La dernière, ***O*** est une propriété de complétude et n'est, stricto sensu, pas nécessaire. Elle est menacée par les sur-spécifications qui risquent de réduire l'efficacité du programme, allant à l'encontre même de l'utilisation de l'assembleur. Détaillons un peu plus ce que recouvrent ces propriétés :

***B*ien-formée.** *Quelle que soit l'instanciation, les mnémoniques assembleur expansées sont valides.*

Il s'agit ici d'établir que les contraintes déclarées respectent celles de l'architecture cible, par exemple, qu'il n'est pas autorisé de choisir deux accès mémoires `"m"` pour un `mov` x86. Considérée comme moins importante, nous ne développons pas cette propriété dans ce papier – une erreur sera mise en exergue par le processus de compilation, sans conséquence donc à l'exécution.

***U*tilisable.** *Si une valeur de retour – en écriture seule – est utilisée par le C, elle est correctement initialisée par le bloc.*

Cette propriété permet d'assurer que toutes les valeurs de retour sont bien définies.

***E*xhaustive.** *Chaque emplacement – registre ou mémoire – sur lequel le bloc peut agir est correctement décrit dans l'interface.*

Cette propriété s'assure que l'interface est complète, c'est-à-dire que les emplacements lus sont préalablement initialisés ou déclarés en lecture et que les emplacements écrits sont soit des sorties,

soit listés comme *clobbers*. La section 2 illustre un des risques du non-respect de ***E***. L'autre risque provient des optimisations que le compilateur va réaliser en se basant uniquement sur ces déclarations, pouvant fausser les véritables chaînes de dépendances – l'affectation non déclarée d'une variable pourrait, par exemple, être ignorée si le bloc est jugé inutile à tort, ou si sa précédente valeur se voit propagée.

***N*on-ambigüe.** *Quelle que soit l'instanciation, le rôle de chaque opérande est le même.*

À l'instar des macros en C, des problèmes peuvent survenir lors du remplacement des opérandes par leur emplacement en présence d'« aliasing ». Un bloc est ambigu si le compilateur est libre de partager le même registre entre une entrée et une sortie malgré le fait que cette sortie est écrite avant que l'entrée ne soit lue. Deux productions sémantiques seront alors observables suivant qu'il décide ou non de faire ce partage. Une ambiguïté apparaît également si un même objet C est associé à différents registres de sortie. L'ordre de *considération* des sorties n'étant pas défini, la valeur reçue par cet objet dépendra ainsi du choix d'implémentation. L'exemple de la Fig. 3 illustre ce problème : les registres `eax` et `edx` sont tous les deux censés venir écraser la valeur de `x` – sans aucune forme d'avertissement, GCC choisira la valeur d'`edx` (`1`) alors qu'icc préférera celle d'`eax` (`0`).

```
int f () { int x; __asm__ ("" : "=a" (x), "=d" (x) : "0" (0), "1" (1)); return x; }
```

FIGURE 3 – Exemple minimal d'interface ambiguë

***O*ptimale.** *Le compilateur dispose de toutes les libertés pour produire le code le plus efficace.*

Cette propriété s'assure quant à elle que l'interface n'a pas été sur-spécifiée. Pour cela, il est nécessaire que chacun des éléments qui la compose soit utile et justifié et que l'application de modificateurs soit réduite au strict minimum requis par la correction. L'exemple le plus répandu de sur-spécification est l'usage du *clobber* `"memory"` là où des alternatives plus précises permettraient au compilateur de ne pas avoir à oublier toutes ses connaissances sur la mémoire.

4 Implémentation

Notre prototype RUSTINA repose sur l'extraction de la sémantique des mnémoniques proposée pour TINA [6, 7] – outil d'aide à la vérification de code mixte C et assembleur embarqué – à laquelle nous ajoutons une analyse dédiée pour établir des diagnostics automatiques et générer les correctifs correspondants. Les Tables 1 and 2 donnent les propriétés prises en charge par RUSTINA.

TABLE 1 – Aperçu des propriétés

	Garantit[1]	Réfute[1]	Rustine
B	hors-sujet		
U	✓	✓	✓
E	✓	∼	∼
N	✓	×	×
O	×	✓	✓

Contexte. Une première analyse de la fonction C à l'aide de Frama-C [5] permet de déterminer les variables ***U***tiles à la sortie du bloc – afin de distinguer les sorties des temporaires. L'interface du bloc assembleur est ensuite parcourue, les objets du C ainsi que les registres de la section *clobbers* sont étiquetés en fonction des spécifications (*l* – lecture, *e* – écriture). Un diagramme de juxtaposition est ensuite calculé (comme dans la Fig. 2). La production de ce diagramme nécessite par ailleurs une connaissance de l'architecture visée, plus précisément sur les classes de registres qu'elle possède et la façon dont sont peuplées les contraintes.

Extraction. Comme pour TINA, le code est ensuite compilé avec les informations de débogage DWARF [3] – le compilateur ne joue qu'un rôle minime, se contentant de transférer l'assembleur

1. *Garantie* : si l'outil ne rapporte pas d'erreur alors il n'y en a pas – *Réfuté* : les erreurs rapportées sont vraies.

TABLE 2 – Garanties, détections de bug et correctifs apportés par RUSTINA

	Règle	Garantit	Réfute	Rustine
U	Pas de sortie non initialisée	✓	✓	*R*0. promeut la sortie en lecture & écriture `'+'`[0]
E	Pas d'emplacement lu sans permission :			
	– registre libre	✓	✓	hors-sujet[1]
	– registre *clobber* avant écriture	✓	✓	rejet justifié
	– sortie	✓	✓	*R*0. promeut la sortie en lecture & écriture `'+'`[0]
	Pas d'emplacement écrit sans permission :			
	– registre libre	✓	✓	*R*1. ajoute le registre à la liste des *clobbers*
	– registre d'entrée	✓	✓	*R*2. promeut l'entrée en lecture & écriture `'+'`
	Pas d'accès mémoire non déclaré :			
	– via un pointeur d'entrée	✓	×	*R*3. ajoute le *clobber* `"memory"`
	– via un symbole global	✓	✓	*R*4. ajoute une nouvelle entrée de type `"m"`[2]
N	Pas de doublon de sorties par registre :			
	– variable locale	✓	✓	rejet justifié
	– contenu pointé	✓[3]	×	avertissement indicatif
	Pas de lecture d'entrée altérée	✓	×	rejet indicatif
O	Pas d'entrée non lue :			
	– lecture seule	✓	✓	*R*5. retire l'élément de la liste
	– lecture & écriture	✓	✓	*R*6. transforme en sortie en écriture seule
	Pas de sortie ni utilisée ni écrite	✓	✓	*R*5. retire l'élément de la liste
	Pas de registre *clobber* non écrit	✓	✓	*R*5. retire l'élément de la liste
	Pas de *clobber* `"memory"` superflu :			
	– aucun accès mémoire	✓	✓	*R*5. retire l'élément de la liste
	– accès statiquement bornés	×	✓	*R*7. remplace par des entrées de type `"m"`
	Pas de modificateur `'&'` superflu	×	✓	*R*8. retire le modificateur &

[0] Le compilateur avertira l'utilisateur si l'objet n'est pas initialisé dans le C.
[1] À l'exception du contournement de *clobber* par `push`/`pop`, il s'agit de registre machine, d'appel système, etc...
[2] À condition que le symbole corresponde à une variable globale déclarée dans le C.
[3] Garantie mais très incomplète.

embarqué tel quel. Le bloc est ensuite désassemblé à l'aide de BINSEC [2] et les opérandes sont identifiés à l'aide du DWARF [3].

Cependant, pour garantir les propriétés ***EN*** sur l'ensemble des instanciations possibles du bloc, il est nécessaire de s'assurer que chacune des entrées et sorties ait été correctement identifiée. En effet, une erreur d'identification peut se produire au niveau du binaire si un registre, écrit textuellement dans le code ("en dur", comme `ebx` dans la Fig. 1), est unifié à une entrée ou une sortie valide ou si le compilateur superpose une entrée et une sortie. Ce problème est corrigé en comparant le code obtenu à partir d'instances différentes du bloc : si les objets ne sont pas identifiés de façon identique dans les différentes instances, le bloc viole alors la propriété de ***N***on-ambigüité. Afin d'obtenir des instances où l'allocation de registres est différente, nous nous reposons sur le fait qu'en compilant sans optimisation (option `-O0`), l'ordre des registres sélectionnés est directement relié à celui des éléments de l'interface. Il suffit ainsi d'ajouter un élément ou de faire une rotation de ceux présents – sans connaissance de l'architecture – pour différencier ces instances.

Analyse. Le désassemblage précédent permet de retrouver le graphe de flot de contrôle du bloc, que nous analysons par une forme simple d'interprétation abstraite [1] pour calculer l'ensemble

des opérandes et emplacements lus et écrits à chaque instruction, vérifiant en chaque point que les lectures sont valides (entrée, *clobber* initialisé ou sortie initialisée), que les écritures sont autorisées (*clobber* ou sortie) et qu'aucune entrée n'est lue après qu'une sortie *pouvant* partager le même emplacement – d'après le diagramme de juxtaposition (Fig. 2) – n'ait été mise à jour.

Rustine. Si un problème est détecté par l'analyse, l'outil tentera de proposer une modification de l'interface qui corrigera le problème avant de relancer la procédure sur le bloc. La Table 2 liste les correctifs apportés par RUSTINA ($R0-8$). De manière générale, les rustines sont indépendantes ; il est toutefois possible qu'un correctif un peu lâche de ***BUEN*** entre en conflit avec ***O*** – la correction restant privilégiée dans ce cas. Pour l'heure, seule la rustine $R3$ pourrait être optimisée à l'aide de $R7$.

Diagnostic. Il se peut que le problème détecté ne possède pas de correctif unique. Les symptômes sont alors remontés à l'utilisateur, en utilisant les catégories définies dans la Table 2. L'expertise humaine est en effet nécessaire pour déterminer, en fonction du contexte, l'*intention* du code et appliquer la correction adéquate. La lecture d'un registre *clobber* avant que celui-ci ne soit initialisé est un exemple de problème détecté mais non corrigé – il pourrait s'agir d'un défaut algorithmique comme d'un oubli d'initialisation, auquel cas la valeur resterait toujours inconnue.

5 Expérimentation

Nous avons lancé RUSTINA sur les sources de **204** paquets de la distribution Debian 8.11 contenant de l'assembleur embarqué – incluant des projets comme ffmpeg, ALSA, GMP et libyuv. La Table 3 résume les différents problèmes détectés, selon les catégories listées dans la Table 2 et le nombre de correctifs automatiquement générés proposées. Notre outil a permis, *en 3 minutes*, de détecter 1587 problèmes sur **1289** blocs (41% des blocs) et de générer automatiquement des correctifs pour 1477 d'entre eux, c'est-à-dire 93% des cas. Il nous a également guidés vers des problèmes plus subtils qui peuvent, quant à eux, être corrigés à l'aide de l'expertise humaine comme celui de l'exemple en Fig. 1.

TABLE 3 – Étude empirique de RUSTINA sur de l'assembleur x86 (Debian 8.11)

(a) Statistiques générales

Debian	8.11
# projet	204
# blocs	3091
durée totale	**3min**

(b) Cas de non-conformité

		# détectés	# corrigés
U	Sortie non initialisée	0	0
E	Lecture d'un registre (non SIMD) libre	9	N/A
	`push`/`pop` du registre `ebx`	25	0
	Lecture d'un registre SIMD libre	75	N/A
	Écriture d'un registre SIMD non déclaré	63	63
	Écriture du registre de conditions	1199	1199
	Accès mémoire non déclaré	132	132
N	Lecture d'une entrée altérée	1	0
O	Sortie écriture & lecture non lue	2	2
	Clobber `"memory"` superflu	80	80
	Modificateur `&` superflu	1	1

Nous avons ainsi préparé des correctifs pour 3 projets, ffmpeg, udpcast et ALSA, en prenant en compte les rustines proposées par RUSTINA mais aussi, en tenant compte de leur spécificité, en corrigeant d'autres problèmes, qui sont actuellement hors de portée de l'état d'automatisation de notre outil. Nous avons pour objectif prochain de proposer l'intégration de nos correctifs dans les

projets identifiés, une fois le tri fait – l'automatisation de la génération de correctifs s'accompagne d'une phase, bien manuelle, de soumission de ces derniers.

Nous avons également testé notre approche sur 3 projets ARM : ffmpeg, GMP et libyuv. Les blocs sont cependant beaucoup plus petits – généralement une instruction pour GMP – et avec une sémantique plus simple (architecture RISC). Nous n'avons ainsi pas détecté de problème de conformité – c'est pourquoi les détails de ces expérimentations ne figurent pas ici.

Discussion. Malgré les problèmes de conformité, les codes actuels « fonctionnent » en général comme attendu, notamment grâce aux limitations techniques du processus de compilation. Il est cependant possible de montrer que le compilateur est à même de générer des solutions problématiques aux codes insuffisamment contraints. Cela se produit lorsque les blocs erronés sont plongés dans un contexte différent (réutilisation de code), plus favorable au déclenchement d'optimisations agressives du compilateur : *inlining*, vectorisation, etc... Les menaces, aujourd'hui bénignes, sont donc à prendre au sérieux alors que les compilateurs incorporent des optimisations toujours plus poussées.

6 Conclusion

RUSTINA est un analyseur qui permet aux utilisateurs d'assembleur embarqué de vérifier que leurs spécifications et implémentations coïncident. C'est une analyse peu coûteuse et qui permet de s'assurer que les interfaces d'assembleur embarqué décrivent un code qui interagira de manière correcte avec son environnement C, et notamment qui comporte toutes les informations nécessaires au compilateur pour insérer le bloc dans le code émis. RUSTINA a déjà permis de trouver un certain nombre de cas problématiques d'usage d'assembleur en ligne dans la distribution Linux Debian 8.11, et de proposer automatiquement des correctifs pour ces derniers.

Références

[1] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL. ACM, 1977.

[2] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In SANER. IEEE Computer Society, 2016.

[3] DWARF Debugging Information Format Committee. DWARF Debugging Information Format 5, 2017.

[4] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried. Some assembly required - program analysis of embedded system code. In SCAM. IEEE Computer Society, 2008.

[5] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c : A software analysis perspective. Formal Asp. Comput., 27(3) :573–609, 2015.

[6] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet. De l'assembleur sur la ligne? Appelez TInA! In JFLA, 2019.

[7] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet. Get rid of inline assembly through verification-oriented lifting. In ASE. IEEE Computer Society, 2019.

[8] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck. An analysis of x86-64 inline assembly in c programs. In VEE. ACM, 2018.

Gardez votre mémoire fraîche avec Memthol

Pierre Chambart[1], Albin Coquereau[1], and
Jacques-Henri Jourdan[2]

[1] OCamlPro SAS, Gif-sur-Yvette, F-91190
[2] LRI, Université Paris Sud, CNRS, Orsay F-91405

Résumé

Chez OCamlPro, notre pratique d'OCaml a évolué. Alors que nous développions majoritairement des programmes terminant, notre modèle d'affaires s'est élargi, ce qui nous a menés à produire des systèmes conçus pour fonctionner sur de longues durées. Pour ces usages, la manière typique de débogage de OCaml à grand renfort de `Printf` n'est pas adaptée, et ce en particulier pour les fuites mémoire lentes, qui sont difficilement observables hors de la production. Imaginons, au hasard, un serveur maintenant une chaîne de blocs qui fuiterait quelques octets par connexion. En fonctionnement normal, cela serait perdu dans le bruit du glaneur de cellules (GC). Cependant, un attaquant pourrait mettre à profit cette fuite pour perturber le service. Nous proposons une solution intégrée pour découvrir, analyser et traiter cette problématique sur des logiciels déployés, sous la forme d'un outil d'analyse statistique léger permettant de collecter à un coût modeste et en continu les schémas d'allocations et de désallocations. Celui-ci est couplé à un mécanisme de collecte exhaustive du graphe mémoire utilisé quand un problème est identifié. Y est adjointe une interface utilisateur pour attirer l'attention du développeur vers les points clefs et le guider vers la correction.

1 Introduction

Le langage OCaml est doté d'un système de gestion de mémoire automatique par glaneur de cellules. Néanmoins, un tel système ne dispense pas le développeur de se soucier de la mémoire de son programme. Or, il est facile de maintenir des valeurs vivantes par erreur. Par exemple, le développeur peut omettre de nettoyer la structure représentant l'ensemble des connexions actives dans un serveur.

Dans ce cas de figure, la principale difficulté réside dans l'étendue des données à traiter, ce aussi bien en termes de nombre de lignes de code (analyse statique) que de taille du graphe de la mémoire (analyse dynamique). Leur quantité rend l'analyse exhaustive par un humain impossible dans le cadre d'un projet de développement. Les approches pour analyser de tels problèmes ne sont pas suffisamment automatisables, ainsi que nous le détaillerons dans l'état de l'art.

L'analyse statique requiert un investissement qui serait démesuré au regard du bénéfice attendu. Un système capable de déterrer les problèmes qui nous intéressent ici demanderait probablement de représenter complètement la sémantique du langage (en incluant nombre de détails d'implémentation que l'on préfère en général esquiver) pour être capable de suivre précisement le flux des valeurs. Ceci serait très difficilement automatisable, en particulier pour un langage de la complexité de OCaml, et demanderait sans doute à l'utilisateur de fournir un certain nombre de preuves. On ne s'attend pas à ce qu'un dévellopeur soit prêt à celà pour quelque chose qui ne relève pas directement de la correction. En pratique ce qui existe est plutôt capable d'aider le GC à libérer plus tôt des choses qu'il aurait été capable de désalouer[2].

Quant à l'analyse dynamique, elle produit une quantité importante de données à analyser : un programme utilise aisément quelques gigaoctets de mémoire vive. Une telle quantité de données n'est humainement traitable que par le recours à des outils tiers.

Dans cette optique, nous proposons un ensemble d'outils conçus pour assister à l'exploration de ces données. Nous nous sommes attelés à une analyse dynamique des programmes, par conséquent il est souhaitable que les analyses afférentes se déroulent dans des conditions d'usage réel afin de garantir l'obtention de résultats pertinents.

Lorsqu'il est utile de se pencher sur ce type de problème, les ressources techniques et physiques des machines sont proches de leur seuil limite. Donc, par définition, peu de ressources restent disponibles pour être attribuées au surcoût d'un outil de collecte de données. Les outils que nous présentons ont pour vocation de respecter ces besoins et donc d'être légers en termes de consommation des ressources processeur, mémoire vive et mémoire morte.

État de l'art

Ce domaine a bien entendu été défriché par la communauté informatique dans son ensemble, ainsi que nous allons le résumer ci-après.

Dans un système d'exploitation, l'usage mémoire global d'un programme est tracé par des outils qui alertent l'utilisateur à faible coût en cas de problème majeur, mais ne permettent pas d'aller au-delà.

Pour cette étape subséquente, le monde du C développe et utilise depuis des années des outils comme *Valgrind* et en particulier ses plugins *Massif* et *Memchech*[6]. Le premier maintient une liste des zones mémoires allouées par *malloc*. Il est capable de déceler les zones mémoires qui n'ont pas été désallouées ainsi qu'une sous-approximation de celles qui ne sont plus accessibles, tel que le ferait un glaneur de cellules conservatif. Toute valeur dans la mémoire est considérée comme étant un pointeur potentiel. Si un de ces supposé pointeur désigne une addresse effectivemement allouée par malloc, cette zone est considérée comme vivante, ne nécessitant pas d'être désalouée, même si elle ne servira plus.

Le second instrumente le binaire pour tracer toutes les allocations ainsi que les lectures et écritures au cours de l'exécution du programme. Cette combinaison produit les informations que l'on recherche dans le cas du C mais n'est pas applicable à OCaml, car son environnement d'exécution obfusque ce type d'information. Du point de vue de ces outils, les valeurs OCaml font toutes parties d'une unique zone mémoire. De plus, les données y étant régulièrement déplacées par le GC, il est impossible d'associer globalement une adresse à une valeur. Qui plus est, ces outils simulent et instrumentent le code assembleur, ralentissant très fortement (1 à 2 ordres de grandeur) le fonctionnement normal du programme.

Dans l'écosystème OCaml, différents outils ont été créés au fil des années, comme entre autres *OCamlviz*[4], *Memprof*[1, 2] et *Spacetime*. *Ocamlviz* avait pour but d'aider à observer les instrumentations du code effectuées par le développeur. Son léger surcoût lui permettait une analyse en temps réel des programmes, en particulier sur des serveurs. Cela avait pour but de montrer la consommation mémoire ainsi que des compteurs définis par l'utilisateur. Cela suppose cependant que l'utilisateur soupçonne un comportement anormal.

Memprof fut développé par *OCamlPro* dans l'optique de répondre à notre problématique. Il utilisait une version modifiée du compilateur OCaml permettant d'instrumenter les allocations en ajoutant des informations supplémentaires dans les valeurs. Ces informations permettaient de connaître le site d'allocation des valeurs depuis lequel leur type était déduit. L'outil enregistrait régulièrement le graphe mémoire complet. Si le surcoût lié à l'instrumentation était relativement faible ($\sim 10\%$), l'écriture des graphes mémoire entraînait en revanche une période d'arrêt complet du programme. De plus, les modifications apportées au compilateur OCaml étaient lourdes et difficilement maintenables. Memprof était équipé d'une interface graphique

accessible via un navigateur web qui permettait une bonne visualisation de la consommation des données. Les versions supportant Memprof sont maintenant obsolètes.

Spacetime instrumente les allocations de OCaml de manière similaire à Memprof en augmentant la précision de l'information. Là où *Memprof* fournit pour chaque valeur un identifiant d'allocation, *Spacetime* identifie la pile d'appels complète. L'information sur les point d'allocation donne à *Memprof* le type de tête du constructeur créé. On sait par exemple qu'un `::` alloué dans `List.map` est de type α *list*. Une passe d'unification sur le graphe mémoire permet de compléter le type concret de cette variable α. *Spacetime* évite la complexité nécessaire au retypage complet, mais l'information perdue sur les types est compensée par une plus grande précision sur le contexte d'appel. Cela entraîne donc des modifications plus légères au sein du compilateur OCaml. Le principal défaut de cet outil est son coût élevé à l'exécution.

Plus récemment, un nouvel outil, *Statmemprof*[5], a exploré une autre approche par échantillonnage. Cette stratégie permet d'utiliser des techniques de collecte à un coût élevé par donnée produite, puis de l'amortir en ne traçant qu'une faible partie des valeurs du programme. Comme la population suivie est grande (de l'ordre du milliard de valeurs allouées), l'information collectée est statistiquement proche de la donnée réelle. *Statmemprof* permet d'obtenir la pile complète de chaque point d'allocation échantillonné. Les changements sur le compilateur pour cette solution sont légers, *Statmemprof* étant maintenu parallèlement au compilateur depuis plusieurs versions ; le processus d'acceptation dans la distribution officielle est en cours.

La Machine Virtuelle de Java (*MVJ*), celle de *.Net* et des différentes implémentations d'*ECMAScript* fournissent un ensemble d'outils couvrant nos besoins. La compilation de OCaml vers ces différentes plateforme, avec *OCamlJava* ou *Js_of_ocaml*, vient malheureusement avec trop de complication pour que nous puissions exploiter ces outils dans le cas général. Les techniques de collecte de ces machines virtuelles exploitent des informations de type introduite pour permettre la compilation à la volée (en particulier, l'introspection et la décompilation). Comme la compilation de OCaml efface les types, y compris pour ces cibles, ces outils ne sont pas réellement applicables.

Dans ce document, nous allons présenter *Memthol*, une suite d'outils héritière de *Memprof*, construite autour de *Statmemprof*, et permettant de faciliter la collecte et l'analyse de données sur l'utilisation mémoire. Dans un premier temps, nous présenterons ce que nous attendons d'un tel outil par le biais de deux exemples. Nous présenterons ensuite le fonctionnement de *Memthol*.

2 Cas d'usage

Au cours de cette section, nous allons présenter le processus de surveillance, recherche et correction de bogue mémoire par le prisme de deux exemples typiques. Notre premier cas d'utilisation s'intéresse à un programme emblématique du monde OCaml : un outil d'analyse statique. Le second exemple concernera un autre cas étonnemment bien connu des développeurs OCaml, une chaîne de blocs.

Dans le cas de l'analyseur statique, les fuites mémoire ne sont pas un problème à long terme car le programme termine et ses ressources sont désallouées au plus tard à ce moment-là. Néanmoins, réduire l'usage maximal de la mémoire vive peut permettre un meilleur passage à l'échelle.

Par contre, dans le cas d'une chaîne de blocs, une fuite même faible peut s'avérer problématique sur le long terme. Il va de soi qu'une fuite substantielle peut entraîner des pertes de services. Cependant, une fuite très faible est aussi une faille de sécurité, car s'il en maîtrise la

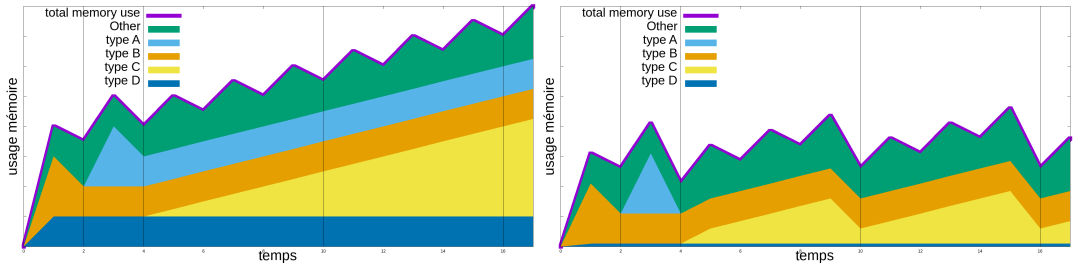


FIGURE 1 – Consommation typique de la mémoire par un analyseur de code et sa version corrigée

vitesse, un attaquant peut saturer la mémoire en noyant le service sous les requêtes, jusqu'à inonder le serveur.

Cas d'un analyseur statique

Le premier graphique de la figure 1 représente schématiquement l'évolution de la consommation mémoire d'un analyseur statique. Ce profil est similaire à ce que l'on a pu observer sur des outils tels que Alt-Ergo[3]. La consommation de mémoire est représentée en fonction d'événements temporels. La première barre verticale correspond à la fin de la phase d'analyse syntaxique, la seconde barre correspond à la fin de la phase de prétraitement (préprocessing). Les barres verticales suivantes correspondent à la fin de chaque analyse de module. Après la phase de prétraitement, chaque pic de consommation mémoire représente les allocations effectuées lors d'une analyse de fonction.

Connaissant la base de code de notre analyseur et son fonctionnement théorique, nous savons que plusieurs problèmes de performance demeurent [1]. Le premier concerne les données allouées durant la phase de prétraitement, qui devraient être désallouées à la fin de cette phase, ici représentée par la courbe `type A`. Le deuxième porte sur le fait que des valeurs sont constamment allouées lors des analyses de fonction. Or, nous nous attendons à ce qu'après chaque fin d'analyse de module, ses données soient libérées. En effet, elles ne sont utiles que pour les fonctions du module couramment étudié. Enfin, le troisième et dernier problème se rapporte à la courbe `type D`. Celle-ci représente des données qui sont allouées lors de l'analyse syntaxique et consomment une partie importante de la mémoire.

Grâce aux données récupérées par notre outil d'analyse du graphe mémoire, nous pouvons résoudre ces différents problèmes. Le second graphique de la figure 1 représente la consommation souhaitée et théorique une fois les anomalies pré-citées résolues. Nous pouvons remarquer que les données `type A` allouées lors de la phase de prétraitement sont maintenant désallouées. Les données `type D`, allouées pendant l'analyse syntaxique et correspondant par exemple à des locations, consomment bien moins de mémoire après optimisation. Concernant les données `type C`, une partie d'entre elles sont maintenant bien désallouées à la fin des analyses des modules.

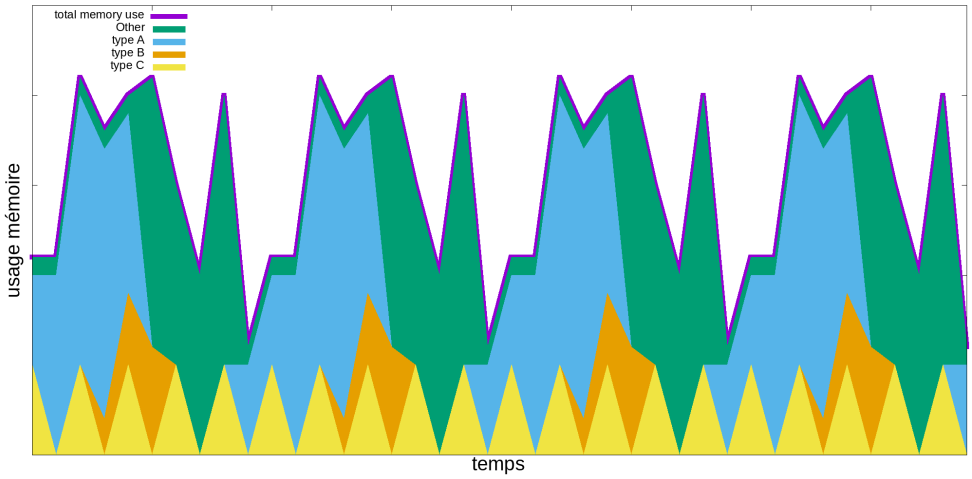


FIGURE 2 – Consommation mémoire par un serveur

Cas d'un serveur

Nous allons maintenant présenter le cas de notre serveur faisant fonctionner une chaîne de blocs. Un autre serveur aurait eu un profil très similaire. Sur la figure 2, nous remarquons que le graphique ne commence pas à 0. Comme nous analysons un serveur, nous devons pouvoir analyser sur des durées longues en affichant, comme ici, une fenêtre de temps et non tout l'historique depuis le lancement du serveur. Le graphique nous montre une certaine périodicité des consommations mémoire. Contrairement à l'exemple précedent, il ne semble pas y avoir de fuite mémoire ou de problème mémoire décelable sur ces courbes.

La figure 3 présente la consommation mémoire de notre serveur avec un filtre restreignant la vue à la durée de vie des données. Nous choisissons par exemple de filtrer les valeurs qui sont encore en vie après dix lancements du glaneur de cellules. Deux types de données se dégagent. Le type `type X` semble stable et ne croît pas, ce qui n'est pas le cas des données de `type Y`. Le nombre d'allocations de ces données augmente peu avec le temps. Cette fuite est difficile à observer dans le bruit des autres données, ainsi que l'illustre la figure 2. Contrairement à l'exemple précédent, nous ne nous basons pas sur nos connaissances du code de l'application pour découvrir une fuite ou un problème de performance.

La possibilité de filtrer les données affichées permet à un utilisateur d'analyser la consommation de son serveur et de signaler une fuite au développeur sans avoir connaissance de la base de code. Pour faciliter le travail de débogage du développeur, l'utilisateur peut lui fournir le filtre en question, ainsi que l'empreinte de la mémoire utilisée pour l'obtention de ces résultats.

3 Memthol

Pour le développement de notre nouvel outil, nous nous sommes appuyés sur les travaux de *statmemprof* pour l'extraction de statistiques portant sur la consommation mémoire d'un pro-

1. De tels corrections ont éffectivement été faite sur Alt-Ergo grâce au information fournie par *Memprof*. https://www.ocamlpro.com/2015/05/18/reduced-memory-allocations-with-ocp-memprof/

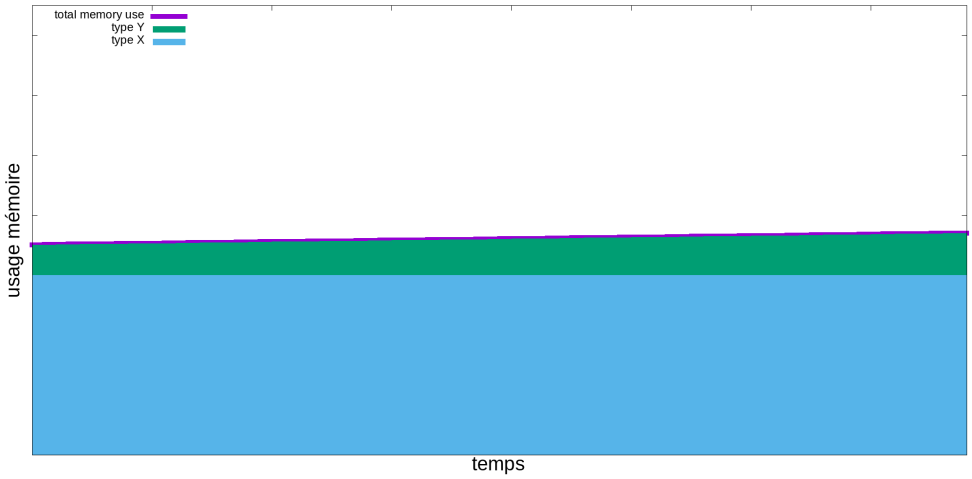


FIGURE 3 – Consommation mémoire d'un serveur en ne gardant que les données ayant survécu à au moins 10 GC allouées après le dixième GC (pour éliminer la phase d'initialisation).

gramme. En ce qui concerne l'instrumentation et l'affichage de ces données, nous nous sommes basés sur les retours utilisateur de *Memprof*. Dans cette section, nous allons présenter notre méthode d'interfaçage avec *statmemprof*, notre technique de collecte des données, et nous conclurons par une rapide présentation de l'interface utilisateur.

Statmemprof

L'outil *Memthol* repose sur une version du compilateur pourvu de *Statmemprof*. Notre bibliothèque permet de définir le ratio d'échantillonnage des valeurs souhaité. De plus, elle offre la possibilité de sélectionner l'intervalle entre deux lancements du glaneur de cellules, ce afin de cibler la phase étudiée.

Nous pouvons ainsi décider de ne collecter les données que tous les dix lancements du glaneur de cellules, et en sus de n'échantillonner qu'une valeur sur mille ou dix mille. *Statmemprof* échantillonant suivant un processus Binomial, le résultat est statistiquement similaire à la consommation réelle du programme, en évitant tout biais éventuel. Utiliser un faible échantillonnage ne pèse que peu sur le fonctionnement normal du programme, ce qui permet d'utiliser Memthol en phase de production.

Statmemprof nous fournit toutes les informations sur le cycle de vie des valeurs : allocation, promotion dans le tas majeur, nettoyage. La connaissance concernant la promotion dans le tas majeur peut se révéler très intéressante pour les programmes qui se trouvent la plupart du temps dans le glaneur de cellules. Avec ces informations, nous sommes en mesure de guider l'effort d'optimisation vers le code qui en a le plus besoin, ce qui est précieux car la promotion peut être particulièrement chère. Si certaines valeurs ont tendance à être promues, il peut donc être bénéfique de tendre à limiter ces promotions, cela en réduisant leur durée de vie, en augmentant la taille du tas mineur, ou encore en évitant de modifier des valeurs dans le tas majeur les contenant (pour éviter le coût de la barrière en écriture).

Des collectes de données sont effectuées au cours du programme selon les ratios choisis. Ces

fichiers ne contiennent que les différences d'allocation vis-à-vis de la collecte précédente. Pour chaque donnée échantillonnée, nous stockons sa pile d'allocation ainsi que sa taille. Une donnée échantillonnée lors de la collecte `n` et désallouée lors de la collecte `m` n'est ainsi pas traitée lors des collectes entre `n` et `m`. Le fait de stocker uniquement les différences d'allocation permet d'avoir des collectes légères, ce qui conserve l'exécution normale du programme, sans le ralentir par l'écriture du fichier de collecte.

Graphe mémoire

Pour collecter le graphe mémoire, les itérateurs du glaneur de cellules sont utilisés pour traverser toutes les racines et les différentes zones mémoire. Cela permet de ne pas modifier le fonctionnement du glaneur et d'externaliser le processus dans une bibliothèque externe. Cette bibliothèque réutilise une fonctionnalitée dévellopée pour *Memprof* qui effectuait un travail semblable pour la collecte du graphe mémoire.

Interface graphique

Memthol est muni d'une interface graphique développée en *Rust*[2]. Cette dernière, à l'image de *Memprof*, est visualisable dans un navigateur web. Elle permet d'afficher simplement la consommation de mémoire en fonction du temps d'exécution du programme analysé. Différents filtres peuvent être ajoutés, tels qu'un filtrage par taille des données, par point d'allocation dans la pile d'appels — permettant de filtrer, par exemple, que les allocations proviennent du module `Set.ml` — et par durée de vie des données, comme présenté dans l'exemple 3.

Exemple et fonctionnement

Reprenons notre cas typique d'application en OCaml, un serveur de chaîne de blocs. En parallèle du fonctionnement du serveur, nous mettons en place un système de surveillance sur le long terme. Pour cela, nous définissons des filtres sur la durée de vie de certaines données ainsi que des alertes lorsque l'une d'elles dépasse un certain seuil. Nous pouvons imaginer vouloir filtrer tout ce qui survit à plus de 100 lancements du glaneur de cellules, et que l'outil nous alerte lorsque la quantité de ces données dépasse 10 Ko.

Il est probable que la première alerte ne soit pas préoccupante et que la donnée ayant déclenché l'alarme soit un faux positif. Si tel est le cas, il faut alors affiner sa recherche en ajoutant de nouveaux filtres. L'opération peut être répétée jusqu'à l'éradication des faux positifs et la constatation que les données observées sont réellement problématiques.

Pour continuer à investiguer, nous lançons une occurrence du serveur localement, ceci afin d'approfondir nos tests. Pour cela, nous ajoutons un filtre spécifique à la valeur problématique. Séparément, sur la machine en production, nous procèdons à une collecte du graphe mémoire complet. Cette opération coûte cher, mais n'est effectuée qu'une fois, et à la demande, contrairement à l'outil *memprof* qui y recourait régulièrement. Nous savons, grâce à notre expérience de *memprof*, que ceci est une information utile lors des opérations de débogage.

Pour que ce graphe mémoire soit utile à l'analyse, les en-têtes des valeurs sont annotés avec les `debuginfo` pour avoir une idée approximative de leur localisation dans le code. L'idée d'annoter les valeurs avec leur type vient, là aussi, du *memprof* original.

Nous comptons utiliser une partie de l'implémentation introduite par *spacetime* pour faire cela sans avoir à modifier le compilateur. Ces annotations sont moins précises et complètes que celles de *memprof* ; en particulier, elles sont insuffisantes pour retrouver systématiquement le

2. Aucune raison particulière, mis à part l'envie du dévellopeur

type. Cependant, elles sont suffisantes pour avoir une idée de la chaîne de valeurs vivantes qui retiennent celles qui nous intéressent. Pour les identifier, il est prévu dans notre feuille de route de faire le lien avec *statmemprof*.

Toutes les valeurs tracées par *statmemprof* sont explicitement marquées dans le graphe mémoire. Ainsi, nous pouvons réappliquer nos filtres précédents pour extraire une faible partie du graphe : les valeurs précédant une valeur que nous avions filtrée. En construisant le graphe inverse, nous obtenons l'ensemble des racines liées aux allocations des valeurs étudiées. Comme ce résultat est raisonnablement petit (il est attendu qu'il dépasse rarement une centaine de valeurs), il est humainement explorable. Nous pouvons donc en extraire beaucoup d'informations sur le comportement effectif du programme.

Cela nous donne un plan précis des modules et du code qu'il va falloir étudier pour déboguer la fuite dans le programme. Par contre, les résultats ne peuvent aider au-delà de ce palier d'alerte ; ce sera ensuite au développeur de comprendre de quelle manière résoudre des problèmes de désallocation, par exemple en se passant de potentiels pointeurs superflus.

Conclusion

Dans ce document, nous avons rappelé les efforts et les besoins de la communauté OCaml autour de l'analyse de consommation de la mémoire. Nous avons ensuite présenté notre prototype d'outil *Memthol* se basant sur les travaux existants, tels que *Memprof* et *Statmemprof*.

Le développement est en cours et nous espérons présenter prochainement une version fonctionnelle de *Memthol* à la communauté. L'étape suivante est probablement d'ajouter une analyse dynamique d'utilité. C'est à dire tracer les accès en lecture et écriture sur chaque valeur. Celà permettrait probablement de fournir de bon filtre afin d'identifier rapidement des fuites, ou même de valeurs mortes.

Références

[1] Çagdas Bozman, Grégoire Henry, Mohamed Iguernelala, Fabrice Le Fessant, and Michel Mauny. ocp-memprof : un profileur mémoire pour OCaml. In David Baelde and Jade Alglave, editors, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, January 2015.

[2] Çağdaş Bozman. *Profilage mémoire d'applications OCaml*. PhD thesis, 2014. Thèse de doctorat dirigée par Mauny, Michel Informatique Palaiseau, Ecole polytechnique 2014.

[3] Sylvain Conchon, Albin Coquereau, Mohamed Iguernelala, and Alain Mebsout. Alt-ergo 2.2. In *Proceedings of the 16th International Workshop on Satisfiability Modulo Theories, SMT 2018*, Oxford, UK, 2018.

[4] Sylvain Conchon, Jean-Christophe Filliâtre, Fabrice Le Fessant, Julien Robert, and Guillaume Von Tokarski. Observation temps-réel de programmes Caml. In Micaela Mayero / Sylvain Conchon, editor, *JFLA (Journées Francophones des Langages Impératifs)*, Studia Informatica Universalis, pages 195–216, Vieux-Port La Ciotat, France, January 2010. INRIA, Hermann Informatique.

[5] Jacques-Henri Jourdan. Statistically profiling memory in OCaml. OCaml Workshop, September 2016.

[6] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

Formalisation de Sémantiques Squelettiques

Louis Noizet and Alan Schmitt

Inria Rennes — Bretagne Atlantique

Résumé

Les sémantiques squelettiques sont une approche récente pour décrire et manipuler des sémantiques opérationnelles de langages de programmation. Une description squelettique peut être utilisée pour prouver la correction d'une sémantique abstraite ou pour générer un interpréteur en OCaml. Nous décrivons dans ce travail comment automatiquement extraire d'une description squelettique une formalisation en Coq de sa sémantique naturelle. Cette formalisation peut ensuite être utilisée pour prouver des propriétés sur la sémantique, que nous illustrons de deux manières : par la preuve qu'un programme calcule bien la fonction factorielle et par la preuve de correction d'un compilateur d'expressions arithmétiques vers une machine à pile.

1 Introduction

La sémantique squelettique [1] est un métalangage pour spécifier la sémantique d'un langage de programmation. Son principe consiste à ne donner que la *structure* de l'évaluation d'un terme (suite d'opérations, récursion, sélection) sans spécifier l'implémentation concrète des opérations de bases, comme l'addition ou la comparaison à zéro. L'approche squelettique permet de découpler la définition d'un langage de définitions ou preuves de propriétés. Par exemple, il est montré dans [1] comment définir une notion de sémantique abstraite indépendemment du langage, qui est correcte par construction si les opérations de base sont correctes. Plus récemment, un générateur d'un interpréteur en OCaml pour un langage décrit en sémantique squelettique a été présenté [3].

Notre travail décrit une formalisation en Coq de sémantiques squelettiques et de leurs interprétations naturelles [5]. Nous utilisons la sémantique squelettique parce que plusieurs langages ont été ou sont en train d'être formalisés dans ce cadre. Après un rappel du cadre formel des sémantiques squelettiques (Section 2), nous décrivons notre formalisation en Coq de ce cadre (Section 3.1) et de l'interprétation naturelle (Section 3.2). Notre outil, dont le code source est disponible en ligne, [1] traduit des descriptions sémantiques en fichiers Coq utilisables pour faire des preuves formelles sur le langage en utilisant cette formalisation. Nous évaluons notre approche au travers de deux exemples : un compilateur certifié de langages simples et la preuve de correction d'un algorithme avec une boucle `while` (Section 4).

2 Sémantique squelettique

Les sémantiques squelettiques ont été développées pour représenter de manière réutilisable et modulaire la structure de sémantiques opérationnelles. L'approche s'appuie sur le fait qu'on peut décomposer la structure d'une sémantique en récursions, branchements et exécutions séquentielles. Le but est ainsi de tirer parti de ces similitudes entre sémantiques pour concevoir un outil modulaire qui puisse servir simultanément à toutes sortes de traitements différents.

$$\frac{\sigma, e \Downarrow \mathit{true} \qquad \sigma, t_1 \Downarrow x_o}{\sigma, \mathit{if}\ e\, t_1\, t_2 \Downarrow x_o} \qquad\qquad \frac{\sigma, e \Downarrow \mathit{false} \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\, t_1\, t_2 \Downarrow x_o}$$

FIGURE 1 – Règles habituelles pour la sémantique concrète d'un *if*

$$\mathit{eval}(x_\sigma, \mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) \coloneqq \left[\mathit{eval}(x_\sigma, x_{t_1})\ ?\triangleright\ x_{f_1}; \begin{pmatrix} \texttt{isTrue}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_2})\ ?\triangleright\ x_o \\ \texttt{isFalse}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_3})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}} \right]_{x_o}$$

FIGURE 2 – Squelette pour la structure *if*

2.1 Exemple

La figure 1 décrit l'interprétation usuelle en sémantique naturelle, ou à grands pas, d'une structure de contrôle conditionnelle. En sémantique squelettique, ce même comportement est décrit dans la figure 2.

Décomposons ce « squelette ».

- $\mathit{eval}(x_\sigma, \mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}) \coloneqq_{\ldots}$: on définit l'évaluation de l'expression *if* $x_{t_1}\ x_{t_2}\ x_{t_3}$ dans l'état x_σ par la fonction d'évaluation *eval*. Cette évaluation est une liste `[ ... ]` d'actions séparées par des `;`.
- $\mathit{eval}(x_\sigma, x_{t_1})\ ?\triangleright\ x_{f_1}$: la première action appelle récursivement l'évaluation de x_{t_1} dans l'état x_σ, et on affecte le résultat à x_{f_1}. On utilise la fonction d'évaluation *eval*, que l'on appelle un *crochet*.
- $\begin{pmatrix} \texttt{isTrue}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_2})\ ?\triangleright\ x_o \\ \texttt{isFalse}\,(x_{f_1})\,; \mathit{eval}(x_\sigma, x_{t_3})\ ?\triangleright\ x_o \end{pmatrix}_{\{x_o\}}$: cette deuxième action est un *branchement*. On évalue chaque ligne séparément, puis on affecte à la variable x_o le résultat de l'évaluation (la manière de calculer le résultat global à partir du résultat de chaque branche dépend de l'interprétation choisie, comme détaillé en Section 2.3). Le $\{x_o\}$ en indice du branchement désigne l'ensemble des valeurs qui sont définies et donc utilisables à l'issue du branchement.
- $\texttt{isTrue}(x_{f_1})$: on calcule si x_{f_1} est vrai. Les primitives du type `isTrue` et `isFalse` sont appelées des *filtres*. Leur comportement n'est pas spécifié dans la sémantique. Chaque filtre prend en argument et renvoie en résultat un certain nombre (fini) de valeurs, possiblement aucune, comme c'est le cas ici. Dans ce cas, le filtre agit comme un sélecteur, les branches étant conservées ou jetées en fonction des arguments donnés au filtre.
- $\mathit{eval}(x_\sigma, x_{t_2})\ ?\triangleright\ x_o$: on évalue récursivement le terme x_{t_2}.
- $[]_{x_o}$: le x_o en indice à la fin du squelette est la valeur qui est renvoyée par le squelette.

On peut observer que cette représentation est très calculatoire : à l'exception des branches, dont on ne donne pas l'ordre d'évaluation, on décrit intuitivement un évaluateur pour la sémantique.

On peut ensuite donner plusieurs *interprétations* possibles au squelette en fonction de la

1. https://gitlab.inria.fr/skeletons/necro/tree/JFLA_2020

sémantique que l'on veut obtenir. Par exemple, l'interprétation concrète fournit une sémantique naturelle, et l'interprétation abstraite peut être utilisée pour prouver que des analyses sont correctes. On peut aussi utiliser une sémantique squelettique pour générer un interpréteur (comme le logiciel necro [2]), un analyseur statique, etc.

Les sémantiques squelettiques sont volontairement simples. Par exemple, elles ne fournissent pas de manière primitive de représenter des lieurs. Si on veut modéliser un langage avec lieur, il faut déclarer, typiquement avec des filtres, des opérations de substitution ou des opérations manipulant des environnements. [3] Cette simplicité offre une grande flexibilité dans la manière dont on définit les sémantiques. Nous ne montrons dans ce travail que des sémantiques à grands pas, mais il est aussi possible de définir des sémantiques à petits pas.

2.2 Définition formelle

Une sémantique squelettique se compose de plusieurs déclarations : les termes, les sortes et les règles.

Les sortes des termes se décomposent en *sortes de base* représentant les éléments de base de notre langage (par exemple les littéraux) et les *sortes de programme* qui correspondent aux termes construits, décrits ci-après. Les termes construits sont ceux qui ont un constructeur en tête et qui seront évalués. On déclare également des *sortes de flux* pour les objets qui seront présents à l'interprétation d'un programme, comme des valeurs, un état ou une pile.

Un terme est une variable x_i ou un constructeur appliqué à des termes. Les termes de bases ne sont instantiés que lorsqu'une interprétation est donnée.

$$\textsc{Terme} \quad t ::= x_i \mid c(t..t)$$

Un squelette a la forme $S_{[x_1..x_n]}$ où S est un corps de squelette

$$\begin{aligned} \textsc{Corps} \quad & S ::= [] \mid B; S \\ \textsc{Os} \quad & B ::= h\,(x_1..x_n, t)\ ?\triangleright\ (y_1..y_m) \mid f\,(x_1..x_n)\ ?\triangleright\ (y_1..y_m) \mid (S..S)_V \end{aligned}$$

Un corps de squelette est une succession d'os. Chaque os est un crochet, un filtre ou un branchement. Un crochet $h(x_1..x_n, t)\ ?\triangleright\ (y_1..y_m)$ correspond à l'évaluation récursive de la sémantique du terme t dans l'état $(x_1..x_n)$, et à l'affectation de son résultat aux variables $(y_1..y_m)$. Un filtre $f(x_1..x_n)\ ?\triangleright\ (y_1..y_n)$ examine si les x_i vérifient certaines pré-conditions et le cas échéant, renvoie des valeurs à affecter aux y_i. Un branchement est du type $(S_1..S_n)_V$ où V est un ensemble de variables qui sont définies dans chaque branche et utilisables à la suite de l'exécution du branchement.

Pour tout ensemble E, on note E^* l'ensemble des suites finies d'éléments de E. Formellement, $E^* := \cup_{n\in\mathbb{N}} E^n$. Donner la sémantique squelettique d'un langage correspond à donner :

- l'ensemble Σ_f des sortes de flux, l'ensemble Σ_b des sortes de base et l'ensemble Σ_p des sortes de programme (on pose $\Sigma = \Sigma_f \cup \Sigma_b \cup \Sigma_p$, et on distingue les sortes de termes $\Sigma_t = \Sigma_b \cup \Sigma_p$) ;
- l'ensemble C des constructeurs, et une fonction csort : $C \to \Sigma_t^* \times \Sigma_p$ qui donne les sortes attendues et la sorte de retour (on note csort_1 et csort_2 ses deux projections) ;
- un ensemble F de filtres, et une fonction fsort : $F \to \Sigma^* \times \Sigma^*$, qui donne les sortes attendues et les sortes de retour ;

2. https://gitlab.inria.fr/skeletons/necro/
3. https://gitlab.inria.fr/skeletons/necro/blob/JFLA_2020/test/lambda_rules.txt

— un ensemble H de crochets, et une fonction hsort : $H \to \Sigma_f^* \times \Sigma_p \times \Sigma_f^*$, qui donne les sortes attendues et les sortes de retour (on note hsort_1, hsort_2 et hsort_3 ses projections) ;
— pour chaque crochet $h \in H$, un ensemble de règles de la forme : $h(y_1..y_n, c(x_{t_1}..x_{t_m})) \coloneqq S_{(z_1..z_p)}$ où $c(x_{t_1}..x_{t_m})$ est le terme évalué ($c \in C$ et $\text{csort}_2(c) = \text{hsort}_2(h)$), les $y_1..y_n$ donnent l'état dans lequel on évalue le terme $c(x_{t_1}..x_{t_m})$ et $S_{(z_1..z_p)}$ est un squelette.

La syntaxe telle qu'elle est décrite ci-dessus ne correspond pas tout à fait à celle définie dans l'article [1], car la sémantique squelettique étant une notion très récente, sa syntaxe est encore en cours d'évolution. Par exemple, on manipule maintenant plusieurs crochets alors que l'article initial n'en définissait qu'un. Ainsi, on peut évaluer une expression de plusieurs manières différentes suivant son contexte. Par ailleurs, la syntaxe initiale ne permettait d'avoir qu'une seule variable en entrée et en sortie d'un crochet, alors que l'on s'autorise maintenant à en avoir un nombre arbitraire. Cela évite d'avoir à systématiquement construire et détruire des n-uplets.

Bonne formation La présentation originelle des sémantiques squelettiques proposait une interprétation de *bonne formation*, qui vérifie que les crochets et les filtres utilisent leurs arguments et valeurs de retour de manière cohérente en ce qui concerne leurs sortes. Pour ce faire, la bonne formation doit se rappeler des sortes des variables squelettiques précédemment définies. Notre formalisation utilise une approche légèrement différente, où nous annotons chaque variable squelettique par sa sorte, comme détaillé en Section 3.1.

2.3 Interprétation concrète

On définit ensuite l'interprétation concrète, correspondant à la sémantique naturelle [5], de cette sémantique. On définit donc la sémantique d'un terme avec des arbres de dérivation qui représentent les appels récursifs. L'idée est donc de commencer par les squelettes sans récursion (sans crochet), qui sont les plus simples. Puis on étend au fur et à mesure en ajoutant les interprétations dont les dérivations sont de plus en plus grandes. Pour ce faire, on définit l'interprétation concrète comme le plus petit point fixe d'une fonctionnelle $\mathcal{H}$ qui à un ensemble d'interprétations déjà prouvées (sous forme de quadruplet (crochet, état, terme, résultat)) associe les dérivations qui peuvent utiliser ces interprétations comme prémisses.

Formellement, un terme clos est un terme dans lequel n'intervient aucune variable. Pour chaque sorte de base b correspond un ensemble de termes concrets de base, qui peuvent être utilisée pour construire des termes concrets clos. À chaque sorte de flux s correspond un ensemble concret de valeurs de flux V_s. On notera $V_{(s_1..s_n)} \coloneqq V_{s_1} \times \cdots \times V_{s_n}$. Une valeur est ou bien une valeur de flux, c'est-à-dire un élément d'un V_s, ou bien un terme clos. Par extension, on notera V_t l'ensemble des termes clos de sorte t.

Pour chaque filtre f tel que $\text{fsort}(f) = (s, t)$ où s et t sont des listes de sortes, on définit son interprétation $[\![f]\!] \in V_s \times V_t$. On note $[\![f]\!]\,(v) \Downarrow w$ pour dire que $(v, w) \in [\![f]\!]$. Pour interpréter un squelette, on va exécuter séquentiellement ses os en mettant à jour un environnement E qui contient les valeurs déjà calculées — E est donc une fonction partielle des variables squelettiques vers des valeurs — et on a également besoin d'un ensemble Q qui contient l'ensemble des résultats des dérivations déjà prouvées, sous la forme de quadruplets concrets (h, v, t, w) où $h \in H$, t est un terme clos et v, w sont des listes de valeurs.

À chaque os B correspond une relation $[\![B]\!]$ qui associe la paire d'un environnement et d'un ensemble Q de quadruplets concrets à son environnement mis à jour et au même ensemble Q. À chaque squelette S correspond une relation $[\![S]\!]$ qui relie une paire du même type que ci-dessus à une valeur de sortie. Ces interprétations sont définies telles que :

$$[\![f\,(x_1..x_m)\ ?\triangleright\ (y_1..y_n)]\!]\,(E, Q) \Downarrow (E' = E + y_1 \mapsto v_1..y_n \mapsto v_n, Q)$$

lorsque $[\![f]\!]\,(E(x_1)..E(x_m)) \Downarrow (v_1..v_n)$;

$$[\![h\,(x_1..x_m, t)\ ?\triangleright\ (y_1..y_n)]\!]\,(E,Q) \Downarrow (E' = E + y_1 \mapsto v_1..y_n \mapsto v_n, Q)$$

lorsque $(h, (E(x_1)..E(x_m)), E(t), (v_1..v_n)) \in Q$;

$$\left[\!\!\left[(S_1..S_m)_{\{x_1..x_n\}}\right]\!\!\right]\,(E,Q) \Downarrow (E' = E + x_1 \mapsto v_1..x_n \mapsto v_n, Q)$$

lorsqu'il existe i tel que $\left[\!\!\left[[S_i]_{(x_1..x_m)}\right]\!\!\right]\,(E,Q) \Downarrow (v_1..v_n)$;

$$\left[\!\!\left[[]_{(x_1..x_n)}\right]\!\!\right]\,(E,Q) \Downarrow (v_1..v_n)$$

lorsque $\forall i, E(x_i) = v_i$;

$$\left[\!\!\left[[B_1; ..; B_n]_{(x_1..x_m)}\right]\!\!\right]\,(E,Q) \Downarrow O$$

lorsque $[\![B_1]\!]\,(E,Q) \Downarrow (E',Q')$ et $\left[\!\!\left[[B_2; ..; B_n]_{(x_1..x_m)}\right]\!\!\right]\,(E',Q') \Downarrow O$.

Alors, on définit la fonctionnelle de conséquence immédiate $\mathcal{H}$ de la manière suivante :

$$\mathcal{H}\,(Q) = \left\{ (h, (\sigma_1..\sigma_n)\,, t, (v_1..v_m)) \;\middle|\; \begin{array}{c} t = c\,(t_1..t_p) \wedge \mathrm{csort}_2(c) = \mathrm{hsort}_2(h) \\ (t_1..t_p) : \mathrm{csort}_1(c) \\ h\big(x_{\sigma_1}..x_{\sigma_n}, c\,\big(x_{t_1}..x_{t_p}\big)\big) \coloneqq S_{[y_1..y_m]} \\ (\sigma_1..\sigma_n) : \mathrm{hsort}_1(h) \\ \Sigma = x_{\sigma_1} \mapsto \sigma_1..x_{\sigma_n} \mapsto \sigma_n + x_{t_1} \mapsto t_1..x_{t_p} \mapsto t_p \\ [\![S]\!]\,(\Sigma, Q) \Downarrow \Sigma' \\ \forall i = 1..m,\ \Sigma'\,(y_i) = v_i \end{array} \right\}.$$

Cette fonctionnelle construit un ensemble de quadruplets $(h, (\sigma_1..\sigma_n), t, (v_1..v_m))$ de la manière suivante. Elle extrait le constructeur de tête du terme t, elle vérifie qu'il est compatible avec le crochet considéré et que les sous-termes sont compatibles avec le constructeur, elle trouve la règle associée au crochet et au constructeur, elle vérifie que les valeurs d'entrées $(\sigma_1..\sigma_n)$ sont compatibles avec le crochet, elle évalue le squelette dans un environnement initial et enfin elle valide que les valeurs retournées sont $(v_1..v_m)$.

Ainsi, $\mathcal{H}^n(\emptyset)$ contient l'ensemble des évaluations qui peuvent être obtenues par des dérivations de taille au plus n. L'interprétation concrète $\Downarrow$ est alors définie par $\Downarrow\ = \bigcup_n \mathcal{H}^n(\emptyset)$.

Prenons l'exemple du *if* (figure 2). On veut calculer la sémantique de $eval(\sigma, if\ t_1\, t_2\, t_3)$. On pose $E = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_3} \mapsto t_3$, et on suppose que Q contient déjà $(eval, \sigma, t_1, false)$ et $(eval, \sigma, t_3, v)$. Alors on a :

$$[\![eval\,(x_\sigma, x_{t_1})\ ?\triangleright\ x_{f_1}]\!]\,(E,Q) \Downarrow (E' = E + x_{f_1} \mapsto false, Q).$$

On entre dans une branche donc on fait séparément les deux branches. `isTrue` va jeter `false` et `isFalse` va le conserver : On a

$$[\![\texttt{isFalse}\,(\texttt{false})]\!]\,(E',Q) \Downarrow (E',Q).$$

Puis, on a

$$[\![eval\,(x_\sigma, x_{t_3})\ ?\triangleright\ x_o]\!]\,(E',Q) \Downarrow (E'' = E' + x_o \mapsto v, Q).$$

Alors, on en déduit que

$$\left[\!\!\left[\begin{pmatrix} \texttt{isTrue}\,(x_{f_1})\,;\, eval\,(x_\sigma, x_{t_2})\,?\triangleright\, x_o \\ \texttt{isFalse}\,(x_{f_1})\,;\, eval\,(x_\sigma, x_{t_3})\,?\triangleright\, x_o \end{pmatrix}_{\{x_o\}} \right]\!\!\right] (E', Q) \Downarrow (E'' = E' + x_o \mapsto v, Q).$$

Ainsi, on peut conclure que $(eval, \sigma, \mathit{if}\ t_1\, t_2\, t_3, v) \in \mathcal{H}(Q)$.

3 Formalisation en Coq

La formalisation en Coq d'une sémantique squelettique comporte trois familles de fichiers. Tout d'abord, les fichiers écrits une fois pour toute et décrivant le cadre formel : la définition des squelettes (Section 3.1) et la définition d'une interprétation concrète (Section 3.2). Ensuite, des fichiers générés automatiquement à partir de la description squelettique de la sémantique. Ces fichiers instancient la définition générale des squelettes pour la sémantique considérée. Enfin, des fichiers supplémentaires fournissent l'implémentation des filtres, ou donnent des propriétés sur ceux-ci, en fonction de l'application considérée. Cette troisième famille de fichiers sera décrite en Section 4.

3.1 Squelettes

Ce premier fichier Skeleton.v formalise la syntaxe des sémantiques squelettiques ainsi que la notion de bonne formation.

On suppose donnés tous les éléments de base d'une sémantique squelettique (l'ensemble des sortes Σ, l'ensemble des constructeurs, filtres, crochets et les fonctions csort, fsort et hsort). Alors on donne la définition d'un terme de sorte `s` (on exclut volontairement les termes de base, qui n'existent pas au niveau de la sémantique squelettique, mais qui seront instanciés au niveau des programmes et des interprétations).

```
Inductive skel_var : sort -> Type :=
| skel_var_intro : forall s, string -> skel_var s.

Inductive term : term_sort -> Type :=
| term_sv : forall t, skel_var (Term t) -> term t
| term_constructor : forall (c:constructor),
    list_term (fst (csort c)) -> term (Prgm (snd (csort c)))
with list_term: list term_sort -> Type :=
| nil_term: list_term []
| cons_term: forall a A, term a -> list_term A -> list_term (a::A).
```

Les définitions d'un os et d'un squelette sont exactement les mêmes que dans la formalisation ci-dessus, à l'exception des variables squelettiques qui sont explicitement typées (leur sorte est donnée) :

```
Inductive bone :=
| H : hook -> list typed_skel_var -> typed_term -> list typed_skel_var -> bone
| F : filter -> list typed_skel_var -> list typed_skel_var -> bone
| B : list (list bone) -> list typed_skel_var -> bone.
Definition skeleton := list (bone).
```

Une sémantique squelettique consiste enfin en la donnée d'une liste de règles du type $h(y_1..y_n, c(x_{t_1}..x_{t_n})) \coloneqq S_{(z_1..z_p)}$, modélisées par le uplet $(h, [y_1..y_m], c, [x_{t_1}..x_{t_p}], S, [z_1..z_p])$, où les variables squelettiques sont elles aussi typées.

```
Definition skeletal_semantics :=
  list (hook * list typed_skel_var * constructor * list typed_skel_var *
        skeleton * list typed_skel_var).
```

Par ailleurs, en utilisant le typage des termes, on définit une fonction `well_formed` à valeurs booléennes, qui vérifie qu'une sémantique est bien formée (code). Enfin, on définit un langage comme un enregistrement contenant tous les ensembles et fonctions précédemment cités, la liste représentant la sémantique squelettique, ainsi que la preuve que la sémantique est bien formée (si on donne une sémantique bien formée, le résultat est immédiat par réflexivité). Le générateur vérifie déjà la bonne formation, mais celui-ci étant écrit en OCaml, la vérification en Coq donne des garanties supplémentaires.

```
Record language := mklang
  {  l_cons: Type;
     l_filter: Type;
     l_hook: Type;
     l_base_sort: Type;
     l_flow_sort: Type;
     l_prgm_sort: Type;
     l_base_sort_eq_dec: forall x y: l_base_sort, {x = y} + {x <> y};
     l_flow_sort_eq_dec: forall x y: l_flow_sort, {x = y} + {x <> y};
     l_prgm_sort_eq_dec: forall x y: l_prgm_sort, {x = y} + {x <> y};
     l_csort: l_cons ->
        ( list (term_sort l_base_sort l_prgm_sort) *
          l_prgm_sort);
     l_fsort: l_filter ->
        ( list (sort l_base_sort l_flow_sort l_prgm_sort) *
          list (sort l_base_sort l_flow_sort l_prgm_sort));
     l_hsort: l_hook ->
        ( list l_flow_sort * l_prgm_sort * list l_flow_sort );
     l_semantics: skeletal_semantics l_cons l_filter l_hook l_base_sort
                    l_flow_sort l_prgm_sort l_csort;
     l_semantics_well_formed: well_formed l_cons l_filter l_hook l_base_sort
                                l_flow_sort l_prgm_sort l_base_sort_eq_dec
                                l_flow_sort_eq_dec l_prgm_sort_eq_dec l_csort
                                l_fsort l_hsort l_semantics
                              = true}.
```

Ce type enregistrement est instancié par les fichiers générés automatiquement, qui se contentent de créer un enregistrement de type `language` (voir par exemple While.v). Par conséquent, le typage automatique des variables squelettiques ne requiert pas d'effort supplémentaire, car ces types sont inférés par l'outil de génération.

3.2 Interprétation Concrète

Nous formalisons maintenant l'interprétation concrète d'un squelette. On suppose donnés un langage, et des fonctions qui définissent l'interprétation des sortes de bases et des sortes de

flux :

```
  Variable l: language.
  Variable base_interp: l.(l_base_sort) -> Type.
  Variable flow_interp: l.(l_flow_sort) -> Type.
```

Les termes concrets sont des termes clos, sans variables de termes, qui représentent des termes du langage. Un terme concrete est soit un terme de base, soit un constructeur appliqué à des termes.

```
Inductive cterm :=
| cterm_base : forall (b:l.(l_base_sort)), base_interp b -> cterm
| cterm_constructor : l.(l_cons) -> list cterm -> cterm.
```

On définit aussi les valeurs, et les quadruplets concrets.

```
Inductive value :=
| val_flow : forall f, flow_interp f -> value
| val_cterm : cterm -> value.

Definition concrete_quadruple := (l.(l_hook) * list value * cterm *
                                  list value)%type.
```

On suppose maintenant qu'on a une interprétation des filtres (donc une fonction qui à chaque filtre associe une relation).

```
Variable filter_interp  : l.(l_filter) -> list value -> list value -> Prop.
```

Enfin, on définit l'interprétation des squelettes, des os, et la sémantique concrète. Deux versions existent, dans deux fichiers différents. La première (Concrete.v) utilise l'induction structurelle de Coq pour interpréter les crochets. La seconde (Concrete2.v) suit la définition donnée en Section 2.3 et définit une fonctionnelle de *conséquence immédiate*, l'homologue de la fonction $\mathcal{H}$ définie plus haut, qui transforme un ensemble de quadruplets en un autre ensemble de quadruplets. La sémantique en est alors le plus petit point fixe.

Voici la définition principale de la première version (code).

```
Inductive interp_skel : skeleton -> env -> env -> Prop :=
| i_Cons : forall B S s1 s2 s3,
    interp_bone B s1 s2 ->
    interp_skel S s2 s3 ->
    interp_skel (B::S) s1 s3
| i_Void : forall s, interp_skel [] s s

with interp_bone : bone -> env -> env -> Prop :=
| i_F : forall lv f (f1 : env) (l1 : list typed_skel_var) l2,
    filter_interp_opt f (List.map (find f1) (map skel_name l1)) lv ->
    interp_bone (F f l1 l2) f1 (add_asn f1 (l2) lv)
| i_H : forall (v1 v2: list value) (h: l.(l_hook)) (ct: cterm)
               (t : typed_term) (e:env) (xf1 xf2:list typed_skel_var),
    eval_term e (projT2 t) = Some ct ->
    map (find e) (map skel_name xf1) = map Some v1 ->
```

```
    concrete_semantics (h,v1,ct,v2) ->
    interp_bone (H h xf1 t xf2) e (add_asn e xf2 v2)
| i_B : forall vals Ss Si V e e',
    List.In Si Ss -> interp_skel Si e e' ->
    map (find e') (map (skel_name) V) = map Some vals ->
    interp_bone (B Ss V) e (add_asn e V vals)

with concrete_semantics :concrete_quadruple -> Prop :=
| cs_intro:
    forall (h: l.(l_hook)),
    forall (c: l.(l_cons)) (s xt xo: list typed_skel_var),
    forall (S: skeleton),
    forall lt s1 s2 sigma,
    In (h,s,c,xt,S,xo) (l.(l_semantics)) ->
    interp_skel S (add_asn (add_asn_cterm void_env xt lt) s s1) sigma ->
    unfold_list_option (map (find sigma) (map skel_name xo)) = Some s2 ->
    concrete_semantics (h, s1, cterm_constructor c lt, s2).
```

Le cas intéressant de cette définition est pour le crochet, qui utilise le prédicat `concrete_semantics`, défini en induction mutuelle, pour la récursion.

Voici la définition principale de la seconde version (code).

```
Inductive interp_skel : skeleton -> Cstate -> Cresult -> Prop :=
| i_Cons : forall B S s1 s2 T r,
    interp_bone B (s1,T) s2 ->
    interp_skel S (s2,T) r ->
    interp_skel (B::S) (s1,T) r
| i_Void : forall s t, interp_skel [] (s,t) s

with interp_bone : bone -> Cstate -> Cresult -> Prop :=
| i_F : forall f (f1 : env) tl lv (l1 : list typed_skel_var) l2,
    filter_interp_opt f (List.map (find f1) (map skel_name l1)) lv ->
    interp_bone (F f l1 l2) (f1,tl) (add_asn f1 (l2) lv)
| i_H : forall (h: (l.(l_hook))) (e : env) (T:concrete_quadruple -> Prop)
    (xf1 xf2 : list typed_skel_var) (t:typed_term) (v : list value),
    match unfold_list_option (map (find e) (map skel_name xf1)),
          eval_term e (projT2 t) with
    | Some xf1', Some t' => T (h, xf1', t', v)
    | _, _ => False
    end -> interp_bone (H h xf1 t xf2) (e,T) (add_asn e xf2 v: env)
| i_B : forall Ss Si V e T vals e',
    List.In Si Ss -> interp_skel Si (e,T) e' ->
    map (find e') (map skel_name V) = map Some vals ->
    interp_bone (B Ss V) (e,T) (add_asn e V vals).

Inductive immediate_consequence : (concrete_quadruple -> Prop) ->
                                  concrete_quadruple -> Prop :=
| H_intro :
    forall (h: l.(l_hook)),
```

```
    forall (c: l.(l_cons)) (s xt xo: list typed_skel_var),
    forall (S: skeleton),
    forall lt s1 s2 T sigma,
    In (h,s,c,xt,S,xo) l.(l_semantics) ->
    interp_skel S (add_asn (add_asn_cterm void_env xt lt) s s1, T) sigma ->
    unfold_list_option (map (find sigma) (map skel_name xo)) = Some s2 ->
    immediate_consequence T (h, s1, cterm_constructor c lt, s2).

Fixpoint consequence n :=
  match n with
  | 0 => fun _ => False
  | S m => immediate_consequence (consequence m)
  end.

Definition concrete_semantics (t : concrete_quadruple) : Prop :=
  exists n, consequence n t.
```

Dans cette version, le crochet s'appuie sur l'ensemble de quadruplets passé en argument pour définir la récursion.

L'équivalence entre ces deux versions est immédiate, et est également prouvée (code). Cependant, on a choisi de garder ces deux versions qui offrent chacune des facilités suivant l'usage prévu. En particulier, pour les deux applications proposées ci-dessous, on utilise tantôt la première version (Section 4.3) et tantôt la seconde (Section 4.2).

3.3 Générateur

Le générateur transforme un fichier de syntaxe type necro en un fichier coq qui produit un enregistrement de type `language`.

On appelle donc un analyseur lexical et un analyseur syntaxique de fichiers de sémantique squelettique, puis un typeur qui vérifie la bonne formation des crochets et ajoute des annotations de type. Ces trois outils sont ceux qui ont été définis pour necro, mentionné ci-dessus.

Ensuite, un fichier prérempli est défini (code), et on se contente de générer les parties manquantes avec un ensemble de fonctions de génération (code).

4 Applications

4.1 Exemple de Génération

Prenons l'exemple de l'opérateur conditionnel `if` défini ci-dessus. Sa version définie en sémantique squelettique s'écrit de la manière suivante (code).

```
If (x_t1, x_t2, x_t3) ->
  x_f1  <- expr x_s x_t1;   (* Appel de crochet *)
  x_f1' <- isBool x_f1;     (* Appel de filtre *)
  x_o <- branch             (* Branchement *)
    isTrue x_f1';           (* Appel de filtre *)
    x_o <- stmt x_s x_t2    (* Appel de crochet *)
  or                        (* Deuxième branche *)
    isFalse x_f1';          (* Appel de filtre *)
```

```
  x_o <- stmt x_s x_t3    (* Appel de crochet *)
end;                      (* Fin du branchement *)
x_o                       (* Valeur renvoyée *)
```

Cette définition est traduite automatiquement par notre outil en ce quadruplet (code).

```
(h_stmt, [tsvi (Flow s_state) "x_s"],
 c_If,
 [tsvi (Term (Prgm s_expr)) "x_t1"; tsvi (Term (Prgm s_stmt)) "x_t2";
  tsvi (Term (Prgm s_stmt)) "x_t3"],
 [H h_expr [tsvi (Flow s_state) "x_s"]
           (tterm_sv (svi (Term (Prgm s_expr)) "x_t1"))
           [tsvi (Flow s_value) "x_f1"] ;
    F f_isBool [tsvi (Flow s_value) "x_f1"] [tsvi (Flow s_vbool) "x_f1'"];
    B [[F f_isTrue [tsvi (Flow s_vbool) "x_f1'"] [];
        H h_stmt [tsvi (Flow s_state) "x_s"]
          (tterm_sv (svi (Term (Prgm s_stmt)) "x_t2"))
          [tsvi (Flow s_state) "x_o"]];
       [F f_isFalse [tsvi (Flow s_vbool) "x_f1'"] [];
        H h_stmt [tsvi (Flow s_state) "x_s"]
          (tterm_sv (svi (Term (Prgm s_stmt)) "x_t3"))
          [tsvi (Flow s_state) "x_o"]]]
      [tsvi (Flow s_state) "x_o"]],
    [tsvi (Flow s_state) "x_o"])
```

Il est intéressant de remarquer que les `tsvi` sont les variables squelettiques, annotées par leurs sortes inférées par l'outil.

4.2 Compilation certifiée

Prenons un langage arithmétique et un langage à pile :

```
expr ::=                        stmt ::=
| Const <literal>               | Skip
| Plus <expr> <expr>            | Seq <stmt> <stmt>
| Minus <expr> <expr>           | Push <literal>
| Times <expr> <expr>           | Add | Sub
| Over <expr> <expr>            | Mul | Div
```

On définit leur sémantique squelettique dans les fichiers Arithmetic.sk et Stack.sk. La compilation du premier langage vers le deuxième est définie dans Compile.v.

Le compilateur est prouvé sans choisir d'implémentation concrète pour les sortes et les filtres. Seules sont exigées certaines propriétés sur ceux-ci, que l'on définit comme des axiomes dans le fichier Prelim.v. Par exemple, on demande que les sortes `value` des deux langages soient interprétées de manière similaire, donc qu'il y ait un isomorphisme entre les deux interprétations (code). Par ailleurs, on exige que l'interprétation du filtre `push` soit l'inverse à gauche de celle du filtre `pop` (code). On demande également que les opérations arithmétiques soient interprétées de manière similaire (code) dans les deux langages.

Une fois posés ces axiomes, on peut prouver le résultat suivant (code) qui affirme que les sémantique d'un programme et de sa version compilée sont équivalentes.

```
forall i1 o1 i2 o2, StateRel i1 o1 i2 o2 ->
forall e e', compile e = Some e' ->
  (exists h, concrete_semantics ar bia fia Afilter_interp (h, i1, e, o1)) <->
  exists h, concrete_semantics st bis fis Sfilter_interp (h, i2, e', o2).
```

Pour prouver ce résultat, on choisit le deuxième fichier d'interprétation concrète, qui définit l'induction comme le plus petit point fixe de la fonctionnelle de conséquence immédiate, pour pouvoir prouver le théorème de correction par récurrence sur la hauteur de l'arbre de dérivation (code). Le résultat est assez immédiat.

4.3 Certification de programme

On prend un langage while dont les expressions et déclarations sont définies de la manière suivante.

```
expr ::=                        stmt ::=
| Const <literal>               | Skip
| Var <ident>                   | Assign <ident> <expr>
| Plus <expr> <expr>            | Seq <stmt> <stmt>
| Equal <expr> <expr>           | If <expr> <stmt> <stmt>
| Not <expr>                    | While <expr> <stmt>
```

La sémantique squelettique de ce langage est définie dans le fichier While.sk. On choisit une implémentation concrète du langage. Pour cela, il faut d'abord choisir les termes de base. On prend les chaînes de caractères comme identifiants, et les entiers relatifs comme littéraux. De même, on choisit une implémentation pour les sortes de flot (code). Enfin, on définit également l'interprétation concrète des filtres (code). On écrit dans ce langage un programme pour calculer la factorielle de n (pour alléger les notations, on notera `a;b` pour `Seq a b`).

```
Assign "over" (Const 0);
Assign "fact" (Const 1);
Assign "n" (Const n);
Assign "a" (Const 0);
Assign "i" (Const 0);
While (Not (Equal (Var "n") (Const 0))) (
  Assign "a" (Var "fact");
  Assign "i" (Const 1);
  While (Not (Equal (Var i) (Var n))) (
    Assign "fact" (Plus (Var "fact") (Var "a"));
    Assign "i" (Plus (Var "i") (Const 1))
  );
  Assign "n" (Plus (Var "n") (Const -1))
)
```

La démonstration est ensuite une démonstration classique pour ce genre de résultat. On commence par définir deux lemmes correspondant aux invariants des deux boucles et on montre le résultat suivant par récurrence (code), où `rd_fm_state` accède à une variable de l'état, et où `fact` calcule (mathématiquement) la factorielle.

```
  exists (x:flow_interp s_state), while_semantics
  ( h_stmt,
```

```
    [val_flow s_state init_state],
    fact_prog n,
    [val_flow s_state x] )
  /\ rd_fm_state x v_fact = Some (Int (fact n)).
```

Puis on applique ces deux lemmes pour conclure (code). Cette fois, on choisit l'interprétation concrète utilisant l'induction de Coq, car elle permet plus facilement d'automatiser les calculs. On note par ailleurs que l'initialisation des variables `"a"` et `"i"` a pour unique but de simplifier l'énoncé du lemme de la boucle principale.

5 Travaux Connexes

Les travaux les plus proches des nôtres sont bien entendu la formalisation en Coq de [1]. Les raisons pour lesquelles nous n'avons pas réutilisé ce travail sont les suivantes. Tout d'abord, cette formalisation est très générique : la notion même d'interprétation est formalisée, indépendemment des interprétations concrètes et abstraites. Cela requiert l'introduction de notions supplémentaires qui alourdissent la définition de l'interprétation concrète et qui rendent plus complexe le raisonnement sur celle-ci. De plus, dans un souci de lisibilité du code généré, nous avons préféré nous éloigner de la définition formelle initiale de l'article pour simplifier les définitions. Nous avons montré en section 3.2 l'équivalence de deux interprétations concrètes. Nous pouvons étendre ce résultat pour montrer l'équivalence entre notre formalisation et celle de [1].

Ott [9] est un système permettant de décrire la sémantique de langages de programmation et leurs systèmes de types. Ott utilise une syntaxe accessible définissant ces sémantiques sous forme de règles d'inférences. Il fournit des outils pour traduire ces définitions en interpréteurs exécutables et en spécification dans les assistants de preuve Coq et HOL. Ott propose plus de fonctionnalités que notre approche, comme la gestion des lieurs, mais il est moins flexible dans la génération de code. En particulier, les traits inductifs d'une sémantique sont nécessairement représentés par un inductif en Coq, et la taille de cet inductif est proportionnelle à la taille de la sémantique. L'utilisation d'autres représentations peut être utile, comme nous l'avons vu en Section 4.2, mais également pour faire des preuves sur de grosses sémantiques.

Le système $\mathbb{K}$ [8] définit un cadre formel pour décrire des sémantiques opérationnelles et pour construire des vérificateurs de programmes s'appliquant directement aux définitions sémantiques sans passer par une représentation intermédiaire ou une logique de programme. Les règles sémantiques sont écrites sous la forme de règles de réécriture s'appliquant à un état sémantique. Le système $\mathbb{K}$ a été utilisé pour décrire les sémantiques de plusieurs langages, comme C, Java, ou JavaScript. Le vérificateur de programme s'appuie sur la « matching logic » [7], un système formel permettant de raisonner sur les motifs et les ensembles de termes associés. Cette approche est efficace mais particulièrement complexe : la sémantique de l'outil lui-même est peu documentée [6] et la génération de code pour d'autres outils de preuve n'existe que sous forme de prototype.

Plusieurs outils se focalisent sur la vérification de programmes, comme CFML [2] ou Iris [4], en définissant directement la sémantique du langage considéré dans l'assistant de preuve utilisé. L'utilisation de ces outils requiert la définition au préalable de la sémantique du langage, ce qui peut demander beaucoup d'efforts. Une extension possible de notre approche serait de générer une telle description à partir d'une sémantique squelettique.

6 Conclusion

Nous avons présenté une formalisation des sémantiques squelettiques, de leur sémantique naturelle, ainsi qu'un outil permettant de générer automatiquement du code Coq utilisable avec cette formalisation à partir d'une description textuelle de la sémantique. Nous avons évalué notre approche grâce à deux applications : l'étude de la sémantique naturelle d'une boucle `while`, et la preuve de correction de compilation d'expressions arithmétiques. Ces travaux ont de nombreuses extensions naturelles. Tout d'abord, nous pourrons définir une sémantique abstraite dans le même cadre et prouver sa correction, comme cela a été fait dans [1]. Ensuite, nous pourrons appliquer notre traducteur à d'autres langages et valider que la formalisation générée est bien utilisable pour prouver des propriétés de programmes. Enfin, nous pourrons définir une interprétation prenant la forme d'une *sémantique axiomatique* pour simplifier les preuves de programmes.

Références

[1] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44 :1–31, 2019.

[2] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011.

[3] Nathanaël Courant, Enzo Crance, and Alan Schmitt. Necro : Animating Skeletons. In *ML 2019*, Berlin, Germany, August 2019.

[4] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28 :e20, 2018.

[5] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[6] Liyi Li and Elsa L. Gunter. Isak : A complete semantics of $\mathbb{K}$. Technical report, Computer Science, Univ. of Illinois Urbana-Champaign, 2018.

[7] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4) :1–61, 2017.

[8] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the $\mathbb{K}$ semantic framework. *Journal of Logic and Algebraic Programming*, 79(6) :397–434, 2010.

[9] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott : Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1) :71–122, 2010.

A Generic Framework for Verified Compilers Using Isabelle/HOL's Locales

Martin Desharnais and Stefan Brunthaler

National Cyber Defence Research Institut (CODE), UniBw M, Germany
{martin.desharnais, brunthaler}@unibw.de

Abstract

In this paper, we present a prototype version of a generic framework for formalizing compiler transformations. Our framework leverages Isabelle/HOL's *locales*—a module system for generic formalizations—to abstract over concrete languages and transformations. The framework thus enables us to state common definitions for language semantics, program behaviours, forward and backward simulations, and compilers. We provide generic operations, such as compiler composition, and prove general theorems, resulting in reusable proof components. By demonstrating our idea on a concrete example, we provide evidence of how locales allow reuse and, therefore, enable encapsulation of verification artefacts into modules.

1 Introduction

The mechanically verified formalization of software components has been the subject of much research in the last decades. Especially influential were the CompCert [3] and CakeML [2] projects, which produced realistic compilers from a (large subset of) two real-world programming languages (C99 and Standard ML) to real hardware platforms. These compilers showed both that mechanized verification is feasible and that it has a measurable effect on the dependability of the compiler [6].

We can now observe a shift in perspective, where the idea of mechanically verified software components is becoming a concrete and desirable goal. Formalization projects are increasing in number, but also in size, complexity, and lifetime. There is an analogy to be made with the emergence, in the second half of the 20th century, of software engineering to the point that the term *proof engineering* starts to be used. New and interesting questions now emerge. How to avoid repetition in definitions and proofs? Which concepts can be generalized and reused? How to separate a formalization in independent components, so that multiple people can work in parallel? What should be the interface between such components? How can tooling make proof engineers more productive? What is a good balance between proof readability and the time required to (mechanically) verify it? etc.

In the case of compiler verification, we have a very well-understood domain, with well-known terminology, that builds on decades of research and empirical experience. But as is the case for a lot of small software prototypes, small-scale formalizations constantly redefined similar abstractions and concepts. This is something we wanted to avoid when we started a small formalization, in Isabelle/HOL [4], of three small stack-based languages implementing different optimizations. Inspired by the concept of modularization in software engineering, we separated the general concepts from the language-dependent parts. We learned about, and made use of, Isabelle's locales to devise a small generic framework[1] for the verification of program transformations.

2 Background

In this section, we start with brief overview of the operational semantics of programming languages and follow with a short introduction to Isabelle's locales.

[1] Available at https://github.com/mdesharnais/framework-VeriComp (commit 16906564).

2.1 Programming Language Semantics

The operational semantics of a programming language can be defined as a transition system representing the execution of a program written in this language. A language $L = \langle S, I, F, \rightarrow \rangle$ is defined by a set S of program states, a set $I \subseteq S$ of initial states, a set $F \subseteq S$ of final states, and a transition relation $\rightarrow \subseteq S \times S$. The execution of a program is modelled as a sequence of states $s_1 \rightarrow s_2 \rightarrow ...$ with $s_1 \in I$. An execution is called terminating if there exists a state s_n such that $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n$ and $\nexists s_{n+1}.\ s_n \rightarrow s_{n+1}$, and non-terminating otherwise. A terminating execution is said to be successful if $s_n \in F$ and to go wrong otherwise. These execution behaviours are usually called the program's behaviour and written $s \Downarrow b$.

The compiler from a language L_1 to L_2 is a partial function $\mathcal{C} : S_1 \rightharpoonup S_2$, which maps a program $s \in I_1$ to $\mathcal{C}(s) \in I_2$.

Two programs s and c are said to be equivalent if they exhibit the same behaviour, i.e. $\forall b, s \Downarrow b \iff c \Downarrow b$. This can be established using a bisimulation [5]: the conjunction of a backward and a forward simulation. Consider a binary relation $\approx$, between program states, expressing that two states are to be considered equivalent for a given use case. This relation is called a simulation whenever $\forall s\ s'\ c, s \approx c \wedge s \rightarrow s' \Longrightarrow \exists c', s' \approx c' \wedge c \rightarrow c'$. A backward simulation, thus, shows that every behaviour of the compiled program is also a behaviour of the source program, i.e. the compilation is correct (sound). A forward simulation shows that every behaviour of the source program can be achieved by the compiled program, i.e. the compilation is complete.

2.2 Isabelle's locales

Locales are an Isabelle construct to define parametric theories [1]. They are based on the concept of proof contexts. A theorem of the form

$$\bigwedge p_1\ p_2\ \ldots\ p_n.\ A_1 \Longrightarrow A_2 \Longrightarrow \cdots \Longrightarrow A_m \Longrightarrow C$$

has a set $\{p_1, p_2, \ldots, p_n\}$ of parameters, a set $\{A_1, A_2, \ldots, A_m\}$ of assumptions, and proves a conclusion *C*. Taken together, the sets of parameters and assumptions is called the proof context. Locales enable the user to define a named proof context and reuse it for multiple conclusions, thus avoiding having to repeat its components in every theorem. Consider for example a formalization of monoids.

```
locale monoid =
 fixes
  f :: 'a ⇒ 'a ⇒ 'a (infix ·) and
  e :: 'a
 assumes
  associativity: x · (y · z) = (x · y) · z and
  left-identity: e · x = x and
  right-identity: x · e = x

context monoid begin
 primrec pow :: nat ⇒ 'a ⇒ 'a where
  pow 0 x = e |
  pow (Suc n) x = x · pow n x
 lemma pow-add:
  pow (n + m) x = pow n x · pow m x
  proof . . . qed
end
```

The *monoid* locale consists of a sequence of parameters, introduced by the **fixes** keyword, and a sequence of assumptions, introduced by the **assumes** keyword. When working in a locale context—introduced by the **context** command or directly following a locale definition—new definitions and theorems can be derived from the locale parameters, its assumptions, and previous derived terms and theorems.

The automatically introduced *locale predicate monoid* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow bool$ identifies locale interpretations, i.e. parameters for which the assumptions hold. We can see that locale contexts really just are syntactic sugar for manual contexts by inspecting the theorem *monoid.pow-add* from outside the locale context.

$$monoid\ f\ e \Longrightarrow pow\ f\ e\ (n + m)\ x = f\ (pow\ f\ e\ n\ x)\ (pow\ f\ e\ m\ x)$$

Locales can also be extended by more parameters and assumptions.

locale *monoid-homomorphisms* = M_f: *monoid f* e_f + M_g: *monoid g* e_g
for $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infix** $\cdot$) **and** e_f **and** $g :: 'b \Rightarrow 'b \Rightarrow 'b$ (**infix** $\diamond$) **and** e_g +
fixes *map* $:: 'a \Rightarrow 'b$
assumes
map-distributive: *map* $(x \cdot y) =$ *map* $x \diamond$ *map* y **and**
map-identity: *map* $e_f = e_g$

The *monoid-homomorphisms* locale extends two instances of *monoid*, fixes a projection function *map* between their underlying types, and states two assumptions on its interaction with the two monoid structures. The extended locale contexts are named, so that elements can be accessed, e.g. by writing M_f*.left-identity*. The sequence of parameters required by the extended locales are introduced by the **for** keyword.

Finally, locales can be interpreted, with the **interpretation** command, by providing values for the parameters and proving that the assumptions hold.

interpretation *monoid-nat-addition*: *monoid* $(+)$ $(0 :: nat)$
proof — Proof that the assumptions hold **qed**

Following interpretation, all derived definitions and theorems, specialized for the provided arguments, are available in the *monoid_nat_addition* namespace.

3 The Design of the Framework

The framework has three main components: some abstract definitions of languages and compilers using locales, a generic definition of program behaviour, and some composition operations over simulations and compilers.

3.1 Locales

The definition of programming languages is separated into two parts: an abstract semantics and a concrete program representation.

locale *semantics* =
fixes *step* $:: 'state \Rightarrow 'state \Rightarrow bool$ **and** *final* $:: 'state \Rightarrow bool$
assumes *final-finished*: *final s* $\Longrightarrow$ *finished step s*

locale *language* = *semantics step final*
for *step* $:: 'state \Rightarrow 'state \Rightarrow bool$ **and** *final* $:: 'state \Rightarrow bool$ +
fixes *load* $:: 'prog \Rightarrow 'state$

The *semantics* locale represents the semantics as an abstract machine. It is expressed by a transition system with a transition relation *step*—usually written as an infix $(\rightarrow)$ arrow—and final states *final*. The *language* locale represents the concrete program representation (type variable *'prog*), which can be transformed into a program state (type variable *'state*) by the *load* function. The set of initial states of the transition system is implicitly defined by the codomain of *load*.

locale *backward-simulation* = *L1*: *semantics step1 final1* + *L2*: *semantics step2 final2* + *well-founded* $(\sqsubset)$
for
step1 $:: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and** *step2* $:: 'state2 \Rightarrow 'state2 \Rightarrow bool$ **and**
final1 $:: 'state1 \Rightarrow bool$ **and** *final2* $:: 'state2 \Rightarrow bool$ **and**
order $:: 'index \Rightarrow 'index \Rightarrow bool$ (**infix** $\sqsubset$) +
fixes *match* $:: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$

assumes
match-final: *match i* s_1 $s_2 \Longrightarrow$ *final2* $s_2 \Longrightarrow$ *final1* s_1 **and**
simulation: *match i* s_1 $s_2 \Longrightarrow$ $s_2 \rightarrow_2 s_2' \Longrightarrow$
$(\exists\, i'\, s_1'.\ s_1 \rightarrow_1{}^+ s_1' \wedge$ *match i'* s_1' $s_2') \vee (\exists\, i'.$ *match i'* s_1 $s_2' \wedge i' \sqsubset i)$

A simulation is defined between two semantics *L1* and *L2*. A *match* predicate expresses that two states from *L1* and *L2* are equivalent. The *match* predicate is also parameterized with an ordering used to avoid stuttering.

The only two assumptions of a backward simulation are that a final state in *L2* will also be a final in *L1*, and that a step in *L2* will either represent a non-empty sequence of steps in *L1*—the $(\rightarrow_1{}^+)$ relation is the transitive closure of the $(\rightarrow_1)$ relation—or will result in an equivalent state. Stuttering is ruled out by the requirement that the index on the *match* predicate decreases with respect to the well-founded $(\sqsubset)$ ordering.

locale *compiler* =
L1: *language step1 final1 load1* + *L2*: *language step2 final2 load2* +
backward-simulation step1 step2 final1 final2 order match
for
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *step2* :: *'state2* ⇒ *'state2* ⇒ *bool* **and**
final1 :: *'state1* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
load1 :: *'prog1* ⇒ *'state1* **and** *load2* :: *'prog2* ⇒ *'state2* **and**
order :: *'index* ⇒ *'index* ⇒ *bool* **and**
match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* +
fixes *compile* :: *'prog1* ⇒ *'prog2 option*
assumes *compile-load*: *compile* p_1 = *Some* $p_2 \Longrightarrow \exists\, i.$ *match i* (*load1* p_1) (*load2* p_2)

The *compiler* locale relates two languages, *L1* and *L2*, by a backward simulation and provides a *compile* partial function from a concrete program in *L1* to a concrete program in *L2*. The only assumption is that a successful compilation results in a program which, when loaded, is equivalent to the loaded initial program.

3.2 Behaviours

We define a generic datatype to encode three broad execution behaviours: successful termination (*Terminates*), non-terminating execution (*Diverges*), and going wrong (*Goes-wrong*).

datatype *'state behaviour* = *Terminates 'state* | *Diverges* | *Goes-wrong 'state*

Terminating behaviours are annotated with the last state of the execution in order to compare the result of two executions with the *rel-behaviour* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a behaviour* ⇒ *'b behaviour* ⇒ *bool* relation.

$$\begin{aligned} f\, s_1\, s_2 \Longrightarrow\ & \textit{rel-behaviour}\, f\ (\textit{Terminates}\ s_1)\ (\textit{Terminates}\ s_2) \\ & \textit{rel-behaviour}\, f\, \textit{Diverges}\, \textit{Diverges} \\ f\, s_1\, s_2 \Longrightarrow\ & \textit{rel-behaviour}\, f\ (\textit{Goes-wrong}\ s_1)\ (\textit{Goes-wrong}\ s_2) \end{aligned}$$

The exact meaning of the three behaviours is defined in the *semantics* locale, where a $(\Downarrow)$ :: *'state* ⇒ *'state behaviour* ⇒ *bool* relation is defined to assign an execution behaviour to a program state. The $(\rightarrow^*)$ relation is the reflexive transitive closure of the $(\rightarrow)$ relation and $(\rightarrow^\infty)$ is its coinductive, infinitely transitive closure. The predicate *finished* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *bool* identifies a state that cannot make a transition.

$$\frac{s_1 \to^* s_2 \qquad \mathit{finished}\ (\to)\ s_2 \qquad \mathit{final}\ s_2}{s_1 \Downarrow \mathit{Terminates}\ s_2}\ \mathit{state\text{-}terminates} \qquad \frac{s_1 \to^\infty}{s_1 \Downarrow \mathit{Diverges}}\ \mathit{state\text{-}diverges}$$

$$\frac{s_1 \to^* s_2 \qquad \mathit{finished}\ (\to)\ s_2 \qquad \neg\ \mathit{final}\ s_2}{s_1 \Downarrow \mathit{Goes\text{-}wrong}\ s_2}\ \mathit{state\text{-}goes\text{-}wrong}$$

Even though the $(\to)$ transition relation in the *semantics* locale need not be deterministic, if it happens to be, then the behaviour of a program becomes deterministic too.

$$\frac{\bigwedge x\ y\ z.\ \dfrac{x \to y \qquad x \to z}{y = z} \qquad s \Downarrow b_1 \qquad s \Downarrow b_2}{b_1 = b_2}$$

The main correctness theorem states that, for any two matching programs, any not wrong behaviour of the later is also a behaviour of the former. In other words, if the compiled program does not crash, then its behaviour, whether it terminates or not, is a also a valid behaviour of the source program. The predicate *is-wrong* :: $'$*state behaviour* $\Rightarrow$ *bool* identifies wrong behaviours.

$$\frac{\mathit{match}\ i\ s_1\ s_2 \qquad s_2 \Downarrow_2 b_2 \qquad \neg\ \mathit{is\text{-}wrong}\ b_2}{\exists\, b_1\ i'.\ s_1 \Downarrow_1 b_1 \wedge \mathit{rel\text{-}behaviour}\ (\mathit{match}\ i')\ b_1\ b_2}$$

Because this theorem is proven in the context of the *backward-simulation* and, thus, only depends on its parameters and assumptions, it is independent of the concrete programming language, and need only be to be proven once. It automatically holds for all interpretations of *backward-simulation*.

As a corollary, the preservation of behaviour can be lifted to the compilation of concrete program representation.

$$\frac{\mathit{compile}\ p_1 = \mathit{Some}\ p_2 \qquad \mathit{load2}\ p_2 \Downarrow_2 b_2 \qquad \neg\ \mathit{is\text{-}wrong}\ b_2}{\exists\, b_1\ i.\ \mathit{load1}\ p_1 \Downarrow_1 b_1 \wedge \mathit{rel\text{-}behaviour}\ (\mathit{match}\ i)\ b_1\ b_2}$$

3.3 Generic Composition of Simulations and Compilers

We define the generic composition of matching functions, $\diamond$ :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow \mathit{bool}) \Rightarrow ('d \Rightarrow 'c \Rightarrow 'e \Rightarrow \mathit{bool}) \Rightarrow 'a \times 'd \Rightarrow 'b \Rightarrow 'e \Rightarrow \mathit{bool}$, and orderings, *lex-prod* :: $('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow ('b \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow \mathit{bool}$, such that the composition of two backward simulations is itself a backward simulation.

$$\frac{\begin{array}{c}\mathit{backward\text{-}simulation}\ (\to_1)\ (\to_2)\ \mathit{final}_1\ \mathit{final}_2\ (\sqsubset_1)\ (\approx_1) \\ \mathit{backward\text{-}simulation}\ (\to_2)\ (\to_3)\ \mathit{final}_2\ \mathit{final}_3\ (\sqsubset_2)\ (\approx_2)\end{array}}{\mathit{backward\text{-}simulation}\ (\to_1)\ (\to_3)\ \mathit{final}_1\ \mathit{final}_3\ (\mathit{lex\text{-}prod}\ ({\sqsubset_1}^+)\ (\sqsubset_2))\ ((\approx_1) \diamond (\approx_2))}$$

We define the generic $(\Leftarrow)$ :: $('a \Rightarrow 'b\ \mathit{option}) \Rightarrow ('c \Rightarrow 'a\ \mathit{option}) \Rightarrow 'c \Rightarrow 'b\ \mathit{option}$ composition operator on compilers, which corresponds to the monadic bind of the *option* type found in a compiler's codomain.

$$(\mathcal{C}_2 \Leftarrow \mathcal{C}_1)\ p \equiv \mathit{Option.bind}\ (\mathcal{C}_1\ p)\ \mathcal{C}_2$$

Its correctness can then be generically proven for any two interpretations of the *compiler* locale.

$$\frac{\begin{array}{c}\mathit{compiler}\ (\to_1)\ (\to_2)\ \mathit{final}_1\ \mathit{final}_2\ \mathit{load}_1\ \mathit{load}_2\ (\sqsubset_1)\ (\approx_1)\ \mathcal{C}_1 \\ \mathit{compiler}\ (\to_2)\ (\to_3)\ \mathit{final}_2\ \mathit{final}_3\ \mathit{load}_2\ \mathit{load}_3\ (\sqsubset_2)\ (\approx_2)\ \mathcal{C}_2\end{array}}{\mathit{compiler}\ (\to_1)\ (\to_3)\ \mathit{final}_1\ \mathit{final}_3\ \mathit{load}_1\ \mathit{load}_3\ (\mathit{lex\text{-}prod}\ ({\sqsubset_1}^+)\ (\sqsubset_2))\ ((\approx_1) \diamond (\approx_2))\ (\mathcal{C}_2 \Leftarrow \mathcal{C}_1)}$$

4 An Instantiation of the Framework

The first programming languages for which we instantiated the framework are three interpreted, stack-based languages. The first one, *Std*, is a standard assembly language with push/pop and load/store instructions, conditional jumps, and unary/binary operations. The second language, *Inca* expands *Std* with inline caching, i.e. operations that are faster for specific operand types but fallback to a generic operation otherwise. The third language, *Ubx*, goes one step further by introducing operations that operate on unboxed operands.

Because we did not want to fix a concrete representation for identifiers or the dynamic environment, we define each language in a locale that abstracts over the relevant types and operations. The *Std* language is defined as follows.

datatype *'id instr* = ...
datatype (*'id*, *'env*) *state* = ...
definition *final* :: (*'id*, *'env*) *state* ⇒ *bool* **where** ...
locale *std* =
 fixes *env-empty* :: *'env* **and** *env-get* :: *'env* ⇒ *'id* ⇒ *'value option* **and** *env-add* :: *'env* ⇒ *'id* ⇒ *'value* ⇒ *'env*
begin
 inductive *step* :: (*'id*, *'env*) *state* ⇒ (*'id*, *'env*) *state* ⇒ *bool* **where** ...
 definition *load* :: *'id instr list* ⇒ (*'id*, *'env*) *state* **where** ...
 lemma *final-finished*: *final s* ⟹ *finished step s* ...
 sublocale *std-sem*: *semantics step final* ...
 sublocale *std-lang*: *language step final load* ...
end

Because locales do not support the definition of new types, the *instr* and *state* datatypes had to be defined in the top-level theory. Moreover, they needed to abstract over the *'id* and *'env* types, which are fixed only inside the locale. The function *final* does not depend on any locale parameter, so it could have been defined at either place. The *step* relation and *load* function, in contrast, depend on the functions to manage the environment and thus need to be defined inside the locale. They instantiate the *instr* and *state* types using the locale's fixed *'id* and *'env*. The lemma *final-finished* can then be stated and proven. Finally, the *semantics* and *language* locales can be interpreted [2], thereby proving that *Std* corresponds to our abstraction of language with a semantics. This also specializes all general results of said locales to this concrete language definition. The two other languages, *Inca* and *Ubx*, follow the same structure.

locale *std-inca-simulation* = *std*: *std env-empty env-get env-add* + *inca*: *inca env-empty env-get env-add*
 for *env-empty* :: *'env* **and** *env-get* :: *'env* ⇒ *'id* ⇒ *'val option* **and** *env-add* :: *'env* ⇒ *'id* ⇒ *'val* ⇒ *'env*
begin
 definition *match* :: *nat* ⇒ (*'id*, *'env*) *Std.state* ⇒ (*'id*, *'env*) *Inca.state* ⇒ *bool* **where** ...
 lemma *match-final*: *match i s1 s2* ⟹ *Inca.final s2* ⟹ *Std.final s1* ...
 lemma *simulation*: *match i* s_1 s_2 ⟹ *inca.step* s_2 s_2' ⟹
  (∃ i' s_1'. *plus std.step* s_1 s_1' ∧ *match* i' s_1' s_2') ∨ (∃ i'. *match* i' s_1 s_2' ∧ $i' < i$) ...
 sublocale *std-ubx-backward-simulation*: *backward-simulation std.step inca.step Std.final Inca.final* (<) *match* ...
end

Proving a backward simulation between *Std* and *Inca* requires to extend the *std* and *inca* locales, define the required *match*, prove the two required *match-final* and *simulation* lemmas, and finally interpret the *backward-simulation* locale.

definition *compile* :: *'id Std.instr list* ⇒ *'id Inca.instr list option* **where** ...
locale *std-inca-compiler* = *sim*: *std-inca-simulation env-empty env-get env-add*
 for *env-empty* :: *'env* **and** *env-get* :: *'env* ⇒ *'id* ⇒ *'value option* **and** *env-add* :: *'env* ⇒ *'id* ⇒ *'value* ⇒ *'env*

[2] The **sublocale** command is a variation of the **interpretation** command.

begin
lemma *compile-load*: *compile p1* = *Some p2* $\Longrightarrow$ *sim.match i* (*std.load p1*) (*inca.load p2*) ...
sublocale *std-to-inca-compiler*:
compiler std.step inca.step Std.final Inca.final std.load inca.load (<) *sim.match compile* ...
end

Defining the compiler and proving its correctness is done in a similar fashion. Because the compilation function depends neither on the dynamic environment nor on the functions defined for the backward simulation, it can be defined in the top-level theory. The *std-inca-compiler* locale merely proves the *compile-load* property and can proceed to interpret the *compiler* locale.

5 Discussion

Using locales as a modularization tool for our generic framework turned out to be elegant at times and frustrating in other cases.

5.1 Strengths of the Approach

Parameters, assumptions, and derived elements are clearly separated. The syntax used to define a locale enables the user to clearly state the parameters and assumptions that are abstracted over. Derived elements such as function definitions and lemmas are clearly separated by being defined later in a locale context. The fact that these extensions can be done at any point following the locale's definition gives a lot of flexibility when structuring the formalization.

It is possible to abstract over multiple types. Locales enable fixed variables to depend on multiple type variables. This makes them more general than type classes, with which they have otherwise a lot in common. While traditional type classes permit to abstract over operations on a given abstract type, locales permit to abstract over both operations on concrete types and multiple abstract types. In fact, type classes in Isabelle/HOL are just syntactic sugar for locales with a single type variable.

It is possible to have multiple interpretations for a given set of type. Because a locale interpretation introduces a new namespace when specializing the derived elements, multiple instantiations are possible for a given set of types. A classical example for such a situation is a partial order over the integers. Using traditional type classes, one has to decide a canonical order that will be associated with the integer type. In order to use an alternate order, one has to define a bijection to an alternative type which instantiate the type class accordingly. As many distinct types and bijections are required as distinct instantiations are wished.

5.2 Weaknesses of the Approach

Parametric types and type aliases cannot be defined in locales. This limitation requires the user to generalize data types to abstract over any type variable fixed in the locale definition and define them outside of the locale. This was e.g. the case for the *instr* and *state* data types of the *Std*, *Inca*, and *Ubx* languages. This generalization is trivial, since a fixed type variable in a locale is akin to a type variable in a data type definition. The burden shows up when referring to the generic type in a type annotation, where it must be explicitly instantiated. Because parametric type aliases are also not supported, the instantiation has to be repeated over and over. As the number of type variables increases, type annotations become complex and hard to read.

When extending existing locales, type annotations on fixed variables are required to name type variables. These variables appear in the **for** section and their types are inferred from their usage in instantiating the extended locales to be extended. Type inference even succeeds in cases where some type variables must be unified between multiple locale instantiations, as is the case in the *compiler* locale. The user must nevertheless provide some type annotations in order to provide the names with which said type variables can be referred to later.

Proving lemmas involving locale predicates have high syntactic overhead. An example of such situation is the *compiler_composition* lemma, where both the two hypotheses and the conclusion are locale predicates of the *compiler* locale. Proving this lemma involves accessing the *language* instance predicates accessible with expressions such as *assms(1)[THEN compiler.axioms(1)]*. The problem with this syntax is twofold: it depends on the order in which the axioms were stated and it does not scale well when the user needs to extract multiple axioms from multiple assumptions. The first problem could be solved by automatically adding lemmas using the name of the extension, e.g. *compiler.L1* would be a synonym of *compiler.axioms(1)* to refer to the first *language* instance predicate of the compiler's definition. The second could be alleviated if unnamed contexts supported extending locales.

Unnamed contexts cannot extend existing locales. They allow to fix variables and state assumptions that are automatically propagated to definitions and lemmas proven in their scope. This is a subset of the possibilities when defining a locale. A next step could be to also allow locale extensions. By propagating the requirement using each locale's instance predication, this would alleviate the syntactical burden of proving lemmas such as *compiler_composition* by allowing a syntax such as *C1.L1*.

References to a locale's fixed variables and derived definitions are syntactically different. When extending locales, as is the case in the *backward_simulation* locale, derived definitions of the two languages are accessible under names such as *L1.behaves* and *L2.behaves*. Fixed parameters, in contrast, are only accessible with their given name *step1* and *step2*. This lack of uniformity means that refactoring a locale, by replacing a fixed parameter by an equivalent derived definition, might require the user to not just adapt all interpretations, but also all derived definitions and theorems to use the prefixed name. Although this would not necessary be a problem in the absence of name clashes, it can be argued that a uniform naming scheme that permits the systematical use the interpretation's prefix would be preferable.

6 Conclusion

We presented the first version of a generic framework for formalizing compilers in Isabelle/HOL. It is based on locales to abstract over the concrete languages and program transformations, provides general definitions for program behaviours and compiler compositions, and generically proves preservation of behaviour. This framework emerged as a side product of our formalization of three stack-based languages that implementing different optimizations. It helped us to emphasis the commonalities between the different theories and to reduce some duplication. Possible future work includes extending the semantics to support traces, exploring how to extract executable programs from such formalizations, and exploring how to simplify or automate repetitive operations such as the composition of multiple compilers.

7 Acknowledgement

This paper is part of the project CONCORDIA, a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

References

[1] C. Ballarin. Locales: A Module System for Mathematical Theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.

[2] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, Jan. 2014.

[3] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[5] D. Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[6] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

Auteurs

Index

Le comité d'organisation des JFLA 2020 remercie chaleureusement ses généreux sponsors.

Nomadic Labs

CEA List

OCamlPro

Tarides

tweag.io

Fondation OCaml

GDR GPL

TrustInSoft